Doctoral Dissertation
Academic Year 2018

A State-Transfer-based Open Framework for Internet of Things Service Composition

Keio University
Graduate School of Media Design

Ruowei Xiao

A Doctoral Dissertation
submitted to Keio University Graduate School of Media Design
in partial fulfillment of the requirements for the degree of
Ph.D of Media Design

Ruowei Xiao

Thesis Advisor:

Associate Professor Kazunori Sugiura	(Principal Advisor)
Professor Akira Kato	(Co-advisor)
Professor Keiko Okawa	(Co-advisor)

Thesis Committee:

Professor Akira Kato	(Principal Advisor)
Professor Keiko Okawa	(Member)
Professor Kai Kunze	(Member)
Senior Assistant Professor Takeshi Sakurada	(Member)

Abstract of Doctoral Dissertation of Academic Year 2018

A State-Transfer-based Open Framework for Internet of Things Service Composition

Category: Science / Engineering

Summary

Current Internet-of-Things (IoT) applications are built upon multiple architectures, standards and platforms, whose heterogeneity leads to domain specific technology solutions that cannot interoperate with each other. It generates a growing need to develop and experiment with technology solutions that break and bridge the barriers.

This research introduces an open IoT development framework that offers general, platform-agnostic development interfaces, and process. It allows IoT researchers and developers to (re-)use and integrate a wider range of IoT and Web services. A Finite State Machine (FSM) model was adopted to provide a uniform service representation as well as an entry point for swift and flexible service composition under Distributed Service Architecture (DSA). Leveraging this open IoT service composition framework, value-added, cross-domain IoT applications and business logic can be developed, deployed, and managed in an on-the-fly manner.

As a typical implementation, a set of web development toolkit named Hyper Sensor Markup Language (HSML) has been developed. Several target domain applications, e.g. multi-source environmental monitoring, open automation systems and etc., have been built. Based on the HSML, the proposed framework has been evaluated by means of user experiment, expert interview and architectural comparison. Results have indicated a better overall performance on expertise requirement, customization cost, reusability and cross-domain interoperability, when compared with other mainstream open IoT service composition frameworks.

The proposed framework has demonstrated its capability to greatly lower down the technical threshold of IoT application development and facilitate fast-prototyping and test over a variety of application domains, including but not limited to smart cities, public environment automation, and precision agriculture.

And going hand in hand with other complementary technologies like semantic web, machine learning and block chain etc., it will hopefully become the primary step towards the equity of future IoT services.

Keywords:

Service Composition, Service Oriented Architecture, IoT Application

Keio University Graduate School of Media Design

Ruowei Xiao

Acknowledgements

The past three and half years have been the most grinding period of my whole life, when I was struggling through endless self-doubt and challenges from both physical and mental aspects. Here, I want to express my gratitude to those people who have generously offered their help during my Ph.D study.

Firstly, I appreciate all the supports from my supervisor Professor Kazurori Sugiura, without him there would be no my research at all. Also I want to thank Professor Akira Kato. As my dissertation committee chair, he always spent the most time and patience to provide concrete academic instructions. Sincere gratitude also for my committee members, Professor Keiko Okawa, Professor Kai Kunze as well as Senior Assistant Professor Takeshi Sakurada from Tokyo University of Agriculture and Technology. Even when extremely occupied, they were always willing to offer inspiring advice and second opinions. Thanks to these kind people, I became more informed about the nature of scientific research: It is all about how to establish a self-consistent theory, how to position and verify it, and most importantly, figure out where your limitation is at.

Secondly, I feel more than grateful to my friends, Wang Dongyu and Yu Xiejing. The three of us first met at the graduate school opening ceremony in September 2012, then became the only three Chinese students in our batch. They were the witnesses to all my trial-and-error within the past six years. Specifically Mr. Wang, he has always been a supportive colleague. And we cooperated in the Omron project, in which I collected a lot of important basic data for this very research. Both of us pursued similar academic goal and also being devotional believers. The only difference may be that he is a Christian, while I am not.

At last, there is another person I have to express specific thanks to. I want to thank Dr. Zhanwei Wu, for being not only my tutor, my colleague, my family, my comrade who fights side by side with me, but also my guide who has led me through those dimmest moments in my life.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Background

The emerging field of compact sensors, actuators and IoT devices offers an unprecedented opportunity for a wide spectrum of applications. Sensors, actuators, IoT devices are greatly featured by their physical entities, and their modes of operation introduce requirements and trade-offs that are very different from traditional systems [1]. The heterogeneity in hardware modalities, sample rates, communication protocols all the way to data schema, further makes the development of applications an excessively complex issue [2].

Currently, most applications are still integrating sensors, actuators and IoT devices through proprietary mechanisms, instead of building upon a well-defined coherent infrastructure [3]. They rely exclusively on vendor-specific platforms and closed technology stack that owned, maintained and used by a single party. This kind of monolithic, ad hoc architecture, often optimized for particular purposes, is able to achieve relatively good real-time performance and high fidelity in specific domains [4], e.g. an industrial automation system or a medical monitoring system.

However, it fails to cope with a more general-purpose, cross-organizational scenarios and dynamic, situational needs. A monolithic code of tightly coupled modules consequently leads to reprogramming efforts to make the network extensible to serve new applications. And once the top application is launched, it is never easy to get any component altered or replaced, which implies limited reusability and inherently low cost-effectiveness, especially in large-scale deployment scenarios.

In recent few decades, emerging protocols for resource-constrained devices like 6LoWPAN, CoAP, EXI and etc., have paved the road for Internet of Things and traditional Web technology stack to converge. Meanwhile, the rapid development of microprocessor technology gives rise to the IoT hardware with richer computing ability and smaller volume, which is playing a more and more important role in

future computational systems. And as high-speed wireless Internet accessibility becomes pervasive, the boundary used to be drew by limited computing resource and communication delay between the modern IoT and the Web has gradually vanished. As a result, we have witnessed an architectural transition took place within IoT application development area within recent decade, shifting from the previously closed, monolithic technology stack to a more open, modular one, as shown in Figure 1.1. Recently, this virtualization and servitization featured tech-

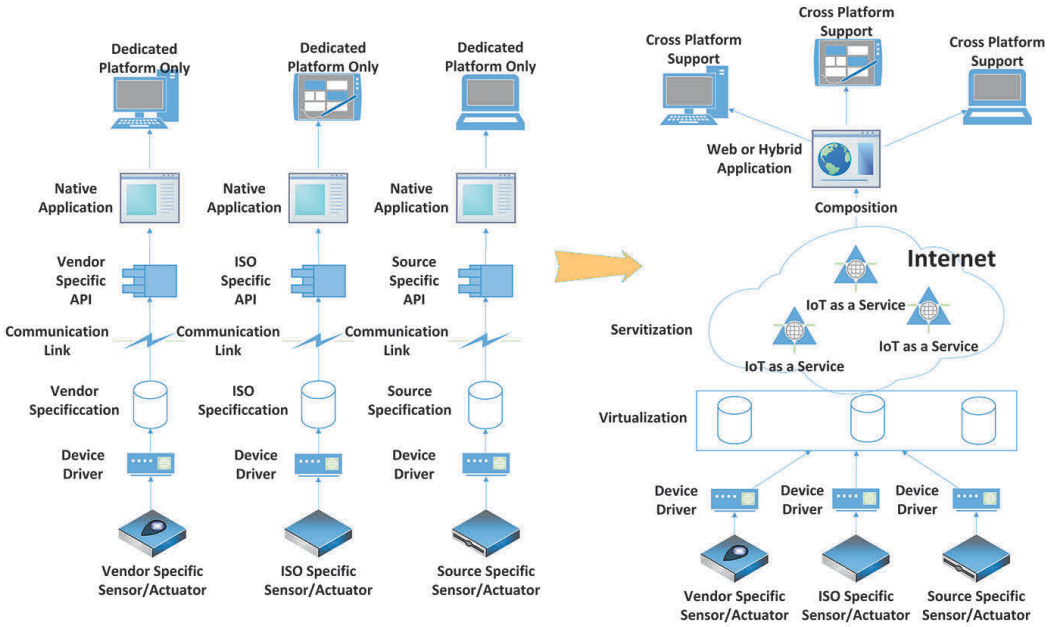


Figure 1.1: From Closed, Monolithic to Open, Modular IoT Application Architecture

nology stack is gaining momentum in both IT, sales and manufacturing industry. Companies like Google, Amazon, GE and Bosch [5] believe that it will help to promote new business model and open innovation by turning heterogeneous, private devices and systems into standard, interoperable services. virtualization refers to the concept that allows the abstraction of physical computing resources into logical units, enabling their efficient usage by multiple independent users [6]. In IoT domain, vitualization can be achieved by different kinds of methods, from container (such as Linux Docker) [7] based to virtual machine based (such as JVM) [8], from deploying lightweight VM (such as node.js) in local or edge device [7] to emulating a whole physical computing environment and related hardware resources on the cloud (such as AWS IoT) [9]. While virtualization simplifies the access and

operation of physical devices by turning them into virtual objects, servitization further provides different virtual objects with uniform, loose-coupling interface so that they can better interact with each other [10]. Servitization is not something new either. It has been successfully practised in enterprise software domain for years, example like ERP [11]. Many service oriented architecture (SOA) and related frameworks have been proposed in the past years, including: language specific ones, e.g. OSGi [12], and web protocol based ones, e.g. SOAP and REST. However, what is the best way to introduce the service oriented architecture into IoT domain is still controversial [13].

In this research, we rely on virtualization and servitization researches to provide necessary lower-layer technical support, since the SOA is believed to be "the only technology stack capable of dealing composite application developments" [14]. Particularly in IoT application area, SOA allows to expose heterogeneous devices and their functionality as independent services with generic service interfaces, while concealing their internal mechanisms and operations. Once sensors, actuators and IoT devices are wrapped up into standard services, i.e. **servitized**, the true capacity can be achieved for the first time through automating customizable tasks by simply aggregating these alike service "blocks" together [15]. However, as many IoT solution providers have already attempted to provide from-device-to-service solutions, such as IBM bluemix, Amazon device shadow, Google Cloud IoT etc, virtualization and servitization research itself is generally considered out of our scope. Though in the following chapters, we will introduce some practical examples to show typical implementation of IoT host services, the framework actually does not depend on any specific virtualization or servitization technology.

Among traditional SOA, Web service is considered to provide more consistent properties with our research aims to lower down the high customization cost and kick-start barriers, and increase the component reusability of current IoT applications, specifically when deployed in large-scale, cross-organizational scenarios. When compared with other services, e.g. Java service, Web service is both language and platform independent, and web technology stack can well support decentralized and distributed computing and is the most widely-adopted technology by various institutions and organizations.

1.2 Research Issues

The open, modular architectural style based on virtualization and servitization have already solved some existing issues in IoT application development domain. Compared with its ancestor, it has concealed vendor-specific APIs/development tools and well absorbed hardware dependency. Off-bottom details, such as protocol-specific communication, are also shielded from the developers to a great extent. However, a few challenges still remain to be addressed:

1. **High expertise requirement and kick-start barriers.** There are several contributors to this issue, among which we put specific emphasis on the complexity of development tools. For novice developers, current IoT application development still rely heavily on specific programming languages, SDKs, and IDEs that require well-trained programming skills and technical expertise. Moreover, how many internal details and operations that developers need to understand in order to use and integrate single IoT service node is another factor that affects the overall learning cost.

2. **High customization cost**. As for customization cost, one of the causes lies in the complicated, usually inconsistent IoT service interfaces. And a lack of efficient service assembly mechanism further leads to large amount of programming and reprogramming work load, and makes constructing business logic or task flow from the bottom up difficult and time consuming. Consequently, it is still prevalent to manually tailor and integrate IoT services to fulfill specific user needs nowadays.

3. **Limited reusability.** Compared with its monolithic ancestor, the open, modular architecture has greatly increased the encapsulation and hence the service reusability. However, the lower level of service encapsulation, which implies a tighter coupling inbetween services, the harder for the service to be reused. And there also remain lots of problems like how to reuse existing functionality and legacy systems in new applications, or how to obtain necessary information of third-party services under distributed service architecture.

4. **Difficult deployment in geographically dispersed, cross-organizational scenarios.** It is very common in application cases like smart city, automation in communal spaces etc., that a multitude of IoT services that

owned and managed by a diversity of organizations and individuals are deployed in a large geographical scale. They are supposed to be networked to provide situational, value-added services collaboratively, which entails an open service architecture that supports resource sharing and discovery.

Targeting the aforementioned issues, this research provides a partial solution based on IoT service composition under distributed service architecture, with specifically focuses on: 1) the interface composability between servitized IoT nodes, 2) the composition mechanism to coordinate services to form an customized task logic, and 3) the user interaction that allows IoT developers to manipulate service composition. As the boundary of our research scope, though the servitization of sensors, actuators and IoT devices is expected to be taken over by service developers, device owners, research communities and part of the manufacturers, typical servitization examples based on mainstream platforms and technologies will also be presented in this research. Also, service discovery and query mechanisms are within our research interests.

To pay specific attentions, issues related to routing, topology managing, and local communication protocols etc., are commonly considered out of scope due to the service homogenization after encapsulating internal technical specifications into web services. Besides, privacy and security are always a concerning issue in regard to networked devices. And unlike traditional computer network, the instability and high mobility of IoT nodes also brings unique challenges in regard to provide reliable quality of services. Security and privacy, fault detection and fault tolerance belong to those issues that varied from case to case, which are not only dictated by underlying web architecture but also by the specific components and composition strategy that IoT developers selected, therefore will not be discussed in details either.

1.3 Research Goal

In a long-term perspective, this research is dedicated to achieve the "**equity of service**" in future IoT field, which envisions that each and every citizen shall have equitable, inclusive accessibility and quality of public IoT infrastructure and resources. Going hand-in-hand with complementary technologies like semantic web, machine learning, and blockchain etc., it is supposed to bring great innovations to the process of IoT application development, deployment and management,

thereby push one step further towards an open, trustable, and autonomous smart society.

Traditional IoT application development is based on multiple architectures, platforms and standards, which usually entails technical expertise and specific knowledge of lower-layer details. One of the consequences are silo systems that cannot interoperate with each other, which has made IoT more like "Internet-connected Things", rather than real "Internet of Things". And the access to IoT welfare was hence monopolized by tech-savvy people.

In this specific research, we attempted to lower down the technical barriers of IoT application development, and provide **cross-domain, platform-agnostic** interoperability among heterogeneous IoT and Web services. To achieve this goal, we proposed an open IoT development framework for research communities, developers and beginners to fast-prototype their IoT applications and test their task logic. It allows developers to (re)use and integrate a wide range of IoT and Web services, which are wrapped into services with unified interfaces regardless of the underlying technical differences. Thus, complicated development procedure is supposed to be simplified and reduced to the assembly and orchestration among selected IoT components, which refers to "**IoT Service Composition**" in this research.

User tests showed that the DSL-based composition tools we provided was beginner-friendly, which generated affordable task load even for novice developers who don't have any previous programming experiences. And architectural evaluation showed that the proposed framework had a better overall performance over customization cost, reusability and cross-domain interoperability, when compared with other mainstream IoT service composition frameworks.

For open access and promotion purpose, we have launched an online open project[1] that could be dated back to the year 2014 and source codes are now available in Github repository[2]. Later, we have collaborated with an automation components and devices manufacturer, Omron Corporation, from 2015 to 2017, during which we have actually implemented the whole framework and deployed in distributed, cross-organizational scenarios.

---

1 `http://www2.kmd.keio.ac.jp/~ruowei.xiao/hsml`

2 `https://github.com/veraxiao/Hyper-Sensor-Markup-Language`

1.4 Research Constraint

To specifically note that, the following issues are generally considered out of the research scope of this dissertation:

1. **Security**. As previously stated, security is always a concerning issue in regard to networked devices. It will greatly relieve developers from being distracted by security issues, if the IoT development framework can provide certain security features. While in our proposed framework, it is feasible to include and integrate external security services, e.g. encryption service, to live up with specific security requirement. But we do not specify any concrete security mechanism within this dissertation.

2. **Privacy**. Similarly, privacy is also a sensitive issue. Introducing external access control mechanism may be a rational solution to ensure that device owners disclose their sensitive private data, e.g. GPS, biophysical data etc., only to their trusted friends and communities, with part of or full access (e.g., readable, referable, editable and full control) according to the trust levels. Though it is considered out of our current research scope, we will further discuss it in future issues.

3. **Real-time Latency**. Due to the IoT service composition approach adopted in proposed framework, the real-time performance of composed applications rely heavily upon the response time of each service node and overall underlying communication infrastructure. It also depends on the actual deployment which varies from case to case. Due to these reasons, we basically do not stress specific attentions on this issue.

1.5 Content Overview

As a whole, this research has proposed an open IoT development framework for composing heterogeneous sensors, actuators and IoT devices into customizable, value-added web applications and business logic. The major contents of this research can be concluded into four unique research results: 1) A platform-independent web development toolkit with HTML-like syntax: Hyper Sensor Markup Language(HSML), 2) Underlying composition mechanism that adopts state-transfer-based service orchestration, 3) A finite-state-machine-based unified resource representation for describing IoT service programming interfaces, namely

StateML, and 4) A full implementation of proposed framework under distributed service architecture, as shown in Figure 1.2.

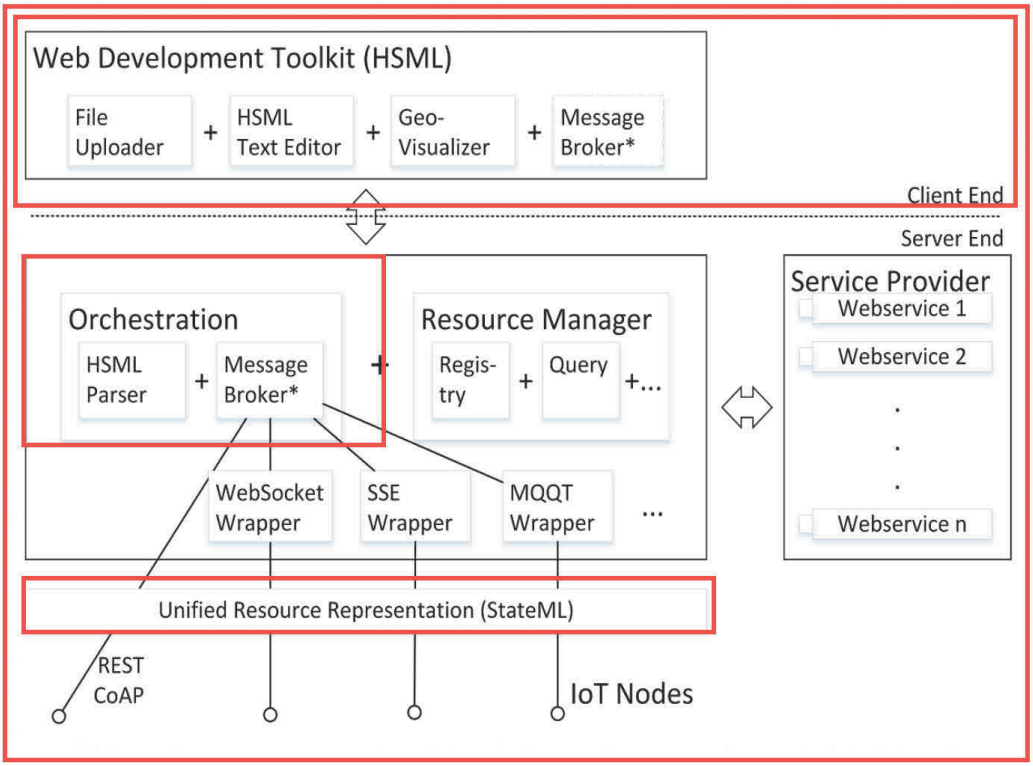


Figure 1.2: Research Contents Overview

To reduce the complexity of development tools, we first provide IoT developers with a set of Web development toolkit, namely HSML. HSML is intentionally devised as a domain-specific language with HTML-like syntax. It allows IoT developers to describe composable IoT service nodes, as well as specify how different services should interrelate in a concise and platform-agnostic manner. Underneath HSML is the proposed composition mechanism, which relies on a central orchestration service to coordinate IoT services. The central orchestration service leverages one or multiple message brokers to receive state messages from previous service node and deliver to the next node according to predefined linking rules. Thus, complicated control logic can be simplified and mapped into state transition chains among the IoT host services that share similar state-based interfaces.

As host services work as an abstract, intermediate layer to interpret vendor-

specific API functions into platform-independent, state-based service interfaces, developers are agnostic about the internal mechanisms inside an host service and of-bottom details below. Instead of ordinary operation-based programming interfaces and remote functions invoking, proposed framework allows developers to specify desired "states" of heterogeneous IoT devices and linking them up using standard Web messaging. Since application logic atop is separated from underlying mechanisms by host services, it can further reduce the reprogramming efforts once hardware get replaces or APIs/drivers altered, and well enhance the reusability of legacy functionality and existing systems.

Developers may concern what states a host service exposes and how the state can be changed, especially when using a remote, third-party IoT resource. This kind of information can be easily expressed by a Finite State Machine (FSM) model. FSM model can represent most of the IoT device behaviors and programming interfaces. To describe the FSM model in a machine-readable format, we also propose StateML, which is a unified resource representation that combines syntax from both Open Geospatial Consortium (OGC)'s Sensor Model Language(SensorML) 2.0 as well as World Web Consortium (W3C)'s State Chart XML (SCXML) standard. It conveys all the necessary information that developers need to access and operate with the resource, including both device-related properties, e.g. data schema, measurement, service address/URI, and the state-based programming interfaces. The FSM model is considered as a key factor to the overall consistency that not only covers the lack of interrelationships among solitary device properties, but also provides general development interfaces to the service orchestration.

The rest of this dissertation is organized as follows: The second chapter systematically concluded existing IoT technology stack layer-by-layer. We then gave the definition of "Open IoT development framework", and introductions of two mainstream genres. Followed with a detailed introduction about Web service architecture, service composition and parallel IoT service composition researches, as it is the target genre that this research anchored on.

Chapter 3 discussed the general approach of IoT service composition that we adopted in this research. The engineering definitions of "state" and "state transfer" were provided. Based on the concepts, the main idea was to encapsulate IoT devices into homogeneous host services that expose unified state-based interfaces, and further compose them into customized task logic by establishing corresponding state-transfer chains. Finite State Machine was adopted to model

IoT host services, and we proposed StateML to explicitly describe FSM-modelled IoT services in a machine-readable format.

Chapter 4 discussed the proposed IoT service composition framework based on the approach. As the prerequisite of our framework, we discussed and provided feasible examples of servitization. As the core of IoT service composition, a state-transfer-based orchestration paradigm was devised. Atop we developed a corresponding development toolkit, Hyper Sensor Markup Language, for IoT developers to establish and manage their service compositions. Detailed syntax, user interface as well as usage samples of HSML were introduced.

In Chapter 5, a comprehensive assessment of proposed framework was carried out to evaluate to what extent our research targets had been achieved. Since expertise requirement and kick-start barriers are compare items closely related to user experience, we hence conducted a user test centering user at beginner level, as well as an expert interview to gain feed backs from veteran users inside the industry. While customization cost, reusability and cross-domain interoperability, were more structural aspects, an architectural comparison together with expert interview were made to systematically review the proposed framework.

Last but not least, Chapter 6 concluded the contributions of this research, briefly analyzed the prospect of proposed IoT service composition technology, as well as discussed the limitations and the remaining issues to be settled in the future.

Chapter 2
Research Background

In this chapter, research background and related researches were presented as a reference to identify the accurate position of our theory within IoT technology spectrum. We first gave a general overview of current IoT technology stack that featured by varying standards and protocols. This complexity impels the wide usage of open IoT development frameworks, which are supposed to provide feasible technical solutions and public-known guidelines for IoT application development. Based on careful literature review, we then roughly divided existing open IoT development frameworks into three genres: 1) Process Virtual Machine based frameworks 2) Domestic Service Hub based frameworks and 3) IoT Service Composition based frameworks. The first two genres were briefly introduced and explained why they were excluded from our solution. Anchored on the last genre, IoT service composition, which originates from traditional Service Oriented Architecture (SOA) and service composition, we hence gave a detailed introduction about mainstream Web service architecture and service composition, along with on-going representative projects and researches within IoT service composition area. And some of these parallel projects were selected as the comparatives of proposed framework in Chapter 5 Evaluation.

2.1 Internet of Things Technology Stack

2.1.1 General Overview

Currently, IoT application development confronts a highly-disperse, complex technology stack, varying intensively from hardware standards to all the way to computing interfaces, as shown in Figure 2.1. In this subsection, we will first provide a general prospect for mainstream IoT technology stacks, which is separated into 4 major blocks from the bottom up, i.e.:

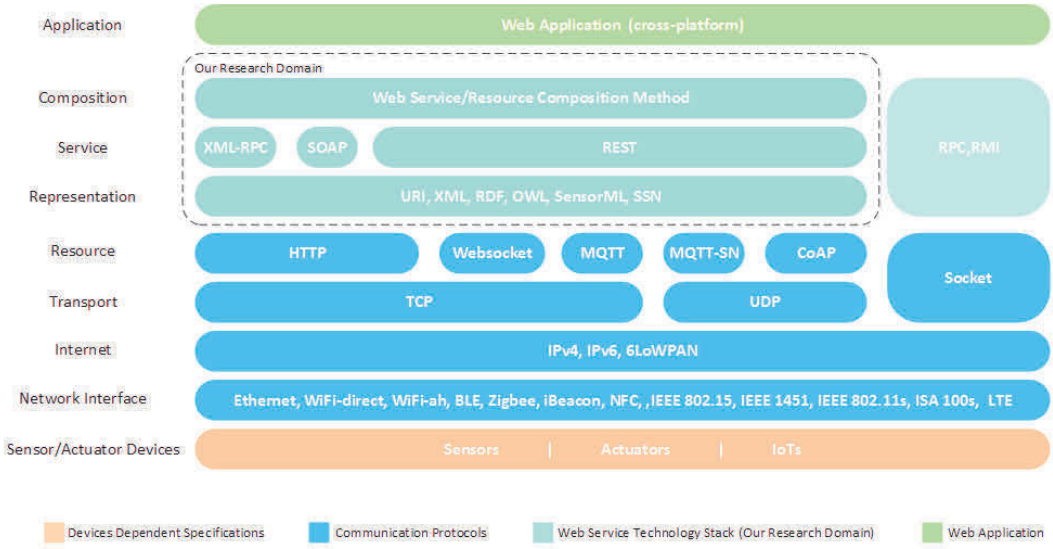


Figure 2.1: Mainstream Web Technology Stack

1. **Sensor/Actuator/IoT Physical Device**, including but not limited to physical devices like sensors, actuators, and IoT systems consisted of sensors and actuators.

2. **Communication Protocol**, roughly divided into network interface protocols, Internet protocols, transport protocols and resource protocols. Together the communication protocol bundle enables the domestic IoT data accessible and exchangeable over the Internet.

3. **Distributed Computing Architecture**, is basically a software middleware for managing data exchange and process synchronization among multiple distributed computing systems. Generally it is considered can be further categorized into three sublayers: resource representation layer, service layer and composition layer.

4. **Web Application**, referring to the actual cross-platform web applications built atop the overall architecture.

A systematic understanding of current IoT technology stack helps us to define our research scope and boundary clearly, hence we will give a layer-by-layer review in the following subsections.

2.1.2 Network Interface Layer

To enable the usage of sensors, actuators and IoT systems (e.g. wireless sensor networks, sensor built-in devices and smart things etc.) in web applications, IoT systems are expected to be addressable and accessible over Internet. At bottom layer of communication protocal stack, mainstream network interface technologies are classified into unconstrained and constrained technologies. The first group includes all the traditional LAN, MAN, and WAN communication technologies, such as Ethernet, WiFi, fiber optic, broadband Power Line Communication (PLC), and cellular technologies such as UMTS and LTE. They are generally characterized by high reliability, low latency, and high transfer rates (order of Mbit/s or higher), and are generally not suitable for peripheral IoT nodes due to their inherent complexity and energy consumption. The constrained physical and link layer technologies are, instead, generally characterized by low energy consumption and relatively low transfer rates, typically smaller than 1 Mbit/s. The more prominent solutions in this category are IEEE 802.15.4, Bluetooth and Bluetooth Low Energy, IEEE 802.11 LowPower, PLC, NFC and RFID. These links usually exhibit long latencies, mainly due to two factors: 1) the intrinsically low transmission rate at the physical layer and 2) the power saving policies implemented by the nodes to save energy, which usually involve duty cycling with short active periods.

2.1.3 Internet Layer

While at the Internet layer of the communication protocol stack, IPv4 is the leading addressing technology supported by Internet hosts. However, IANA, the international organization that assigns IP addresses at a global level, has recently announced the exhaustion of IPv4 address blocks. IoT networks, in turn, are expected to include billions of nodes, each of which shall be (in principle) uniquely addressable. A solution to this problem is offered by the IPv6 standard, which provides a 128-bit address field, thus making it possible to assign a unique IPv6 address to any possible node in the IoT network. While, on the one hand, the huge address space of IPv6 makes it possible to solve the addressing issues in IoT; on the other hand, it introduces overheads that are not compatible with the scarce capabilities of constrained nodes. This problem can be overcome by adopting 6LoWPAN [16], [17], which is an established compression format for IPv6 and UDP headers over low-power constrained networks. A border router, which is a device directly attached to the 6LoWPAN network, transparently performs the

conversion between IPv6 and 6LoWPAN, translating any IPv6 packet intended for a node in the 6LoWPAN network into a packet with 6LoWPAN header compression format, and operating the inverse translation in the opposite direction. While the deployment of a 6LoWPAN border router enables transparent interaction between IoT nodes and any IPv6 host in the Internet, the interaction with IPv4-only hosts remains an issue. More specifically, the problem consists in finding a way to address a specific IPv6 host using an IPv4 address and other meta-data available in the packet. Here are different approaches to achieve this goal.

v4/v6 Port Address Translation (v4/v6 PAT). This method maps arbitrary pairs of IPv4 addresses and TCP/UDP ports into IPv6 addresses and TCP/UDP ports. It resembles the classical Network Address and Port Translation (NAPT) service currently supported in many LANs to provide Internet access to a number of hosts in a private network by sharing a common public IPv4 address, which is used to address the packets over the public Internet. When a packet is returned to the IPv4 common address, the edge router that supports the NATP service will intercept the packet and replace the common IPv4 destination address with the (private) address of the intended receiver, which is determined by looking up in the NATP table the address of the host associated to the specific destination port carried by the packet. The same technique can be used to map multiple IPv6 addresses into a single IPv4 public address, which allows the forwarding of the datagrams in the IPv4 network and its correct management at IPv4-only hosts. The application of this technique requires low complexity and, indeed, port mapping is an established technique for v4/v6 transition. On the other hand, this approach raises a scalability problem, since the number of IPv6 hosts that can be multiplexed into a single IPv4 address is limited by the number of available TCP/UDP ports (65535). Furthermore, this approach requires that the connection be initiated by the IPv6 nodes in order to create the correct entries in the NATP look-up table. Connections starting from the IPv4 cloud can also be realized, but this requires a more complex architecture, with the local DNS placed within the IPv6 network and statically associated to a public IPv4 address in the NATP translation table.

v4/v6 Domain Name Conversion [18]. This method is similar to the technique used to provide virtual hosting service in HTTP 1.1, which makes it possible to support multiple websites on the same web server, sharing the same IPv4 address, by exploiting the information contained in the HTTP Host header to identify the specific web site requested by the user. Similarly, it is possible to

program the DNS servers in such a way that, upon a DNS request for the domain name of an IoT web service, the DNS returns the IPv4 address of an HTTP-CoAP cross proxy to be contacted to access the IoT node. Once addressed by an HTTP request, the proxy requires the resolution of the domain name contained in the HTTP Host header to the IPv6 DNS server, which replies with the IPv6 address that identifies the final IoT node involved in the request. The proxy can then forward the HTTP message to the intended IoT via CoAP.

URI mapping. The Universal Resource Identifier (URI) mapping technique is also described in [18]. This technique involves a particular type of HTTP-CoAP cross proxy, the reverse cross proxy. This proxy behaves as being the final web server to the HTTP/IPv4 client and as the original client to the CoAP/IPv6 web server. Since this machine needs to be placed in a part of the network where IPv6 connectivity is present to allow direct access to the final IoT nodes, IPv4/IPv6 conversion is internally resolved by the applied URI mapping function.

2.1.4 Transport and Resource Layer

Most of the traffic that crosses the Internet layer nowadays is carried at the application layer by HTTP over TCP. However, the verbosity and complexity of native HTTP make it unsuitable for a straight deployment on constrained IoT devices. For such an environment, in fact, the human-readable format of HTTP, which has been one of the reasons of its success in traditional networks, turns out to be a limiting factor due to the large amount of heavily correlated (and, hence, redundant) data. Moreover, HTTP typically relies upon the TCP transport protocol that, however, does not scale well on constrained devices, yielding poor performance for small data flows in lossy environments.

The CoAP protocol [19] overcomes these difficulties by proposing a binary format transported over UDP, handling only the retransmissions strictly required to provide a reliable service. Moreover, CoAP can easily interoperate with HTTP because: 1) it supports the ReST methods of HTTP (GET, PUT, POST, and DELETE), 2) there is a one-to-one correspondence between the response codes of the two protocols, and 3) the CoAP options can support a wide range of HTTP usage scenarios. Even though regular Internet hosts can natively support CoAP to directly talk to IoT devices, the most general and easily interoperable solution requires the deployment of an HTTP-CoAP intermediary, also known as cross proxy that can straightforwardly translate requests/responses between the two protocols, thus enabling transparent interoperation with native HTTP devices

and applications.

Message Queue Telemetry Transport (MQTT) [20] is a client server publish/-subscribe messaging transport protocol atop of TCP/IP protocol stack, which is specifically initiated for constraint environment such as for communication in Machine to Machine (M2M) and Internet of Things (IoT). It is designed to be light-weight, open and able to support reliable message delivery of three different qualities of services. Currently, MQTT and its variation MQTT-SN, which aimed at embedded devices on non TCP/IP networks, developed clients over a variety of platforms and devices, and have established its ecosystem in a nascent stage. Business implementations include Facebook Messenger, Amazon Web Services, EVRYTHNG IoT platform etc. But MQTT requires both servers and clients to store session information to provide reliable, bidirectional connections, which may exceed the storage and computing resources of some constraint devices, and break the stateless principle.

2.1.5 Resource Representation Layer

Atop communication protocol stack, data exchange is typically accompanied by a description of the transferred content by means of semantic representation languages, of which the eXtensible Markup Language (XML) is probably the most common. Nevertheless, the size of XML messages is often too large for the limited capacity of typical IoT devices. Furthermore, the text nature of XML representation makes the parsing of messages by CPU-limited devices more complex compared to the binary formats. For these reasons, the working group of the World Wide Web Consortium (W3C) has proposed the EXI format [21], which makes it possible even for very constrained devices to natively support and generate messages using an open data format compatible with XML. EXI defines two types of encoding, namely schema-less and schema-informed. While the schema-less encoding is generated directly from the XML data and can be decoded by any EXI entity without any prior knowledge about the data, the schema informed encoding assumes that the two EXI processors share an XML Schema before actual encoding and decoding can take place. This shared schema makes it possible to assign numeric identifiers to the XML tags in the schema and build the EXI grammars upon such coding. A general purpose schema-informed EXI processor can be easily integrated even in very constrained devices, enabling them to interpret EXI formats and, hence, making it possible to build multipurpose IoT nodes even out of very constrained devices [22]. Using the schema informed approach,

however, requires additional care in the development of higher layer application, since developers need to define an XML Schema for the messages involved in the application and use EXI processors that support this operating mode. Integration of multiple XML/EXI data sources into an IoT system can be obtained by using the databases typically created and maintained by high-level applications. In fact, IoT applications generally build a database of the nodes controlled by the application and, often, of the data generated by such nodes. The database makes it possible to integrate the data received by any IoT device to provide the specific service the application is built for.

Due to the large variety of sensor protocols and sensor interfaces, most applications are still integrating sensor resources through proprietary mechanisms, instead of building upon a well-defined and established integration layer. This manual bridging between sensor resources and applications leads to extensive adaption effort, and is a key cost factor in large-scale deployment scenarios. This issue has been the driving force for the Open Geospatial Consortium (OGC) to start the Sensor Web Enablement (SWE) initiative [23] back in 2003, which was established to address standardization within sensor web by developing a suite of specifications related to sensors, sensor data models, and sensor Web services that will enable sensors to be accessible and controllable via the Web. The core suite of language and service interface specifications includes the following:

1. **Observations and Measurements (O&M)**. These are standard models and XML schema for encoding archived and real-time observations and measurements from a sensor.

2. **Sensor Model Language (SML)**. These are standard models and XML schema for describing sensors systems and processes. They provide information needed for discovering sensors, locating sensor observations, processing low-level sensor observations, and listing taskable properties.

3. **Transducer Model Language (TML)**. These are standard models and XML schema for describing transducers and supporting real-time streaming of data to and from sensor systems.

4. **Sensor Observation Service (SOS)**. This is the standard Web service interface for requesting, filtering, and retrieving observations and sensor system information. It is also the intermediary between a client and an observation repository or near real-time sensor channel.

5. **Sensor Planning Service (SPS)**. This is the standard Web service interface for requesting user-driven acquisitions and observations. It is also intermediary between a client and a sensor collection management environment.

6. **Sensor Alert Service (SAS)**. This is the standard Web service interface for publishing and subscribing to alerts from sensors.

7. **Web Notification Services (WNS)**. This is the standard Web service interface for asynchronous delivery of messages or alerts from SAS and SPS Web services and other elements of service work flows.

2.2 Open IoT Development Framework

The complexity and diversity of current IoT technology provides plentiful technical options, meanwhile increases technical hurdles and difficulties of development, and consequently introduces the necessity of IoT development frameworks. In a broader sense of computing science, a software or application framework generally refers to "a set of common software routines that provides a foundation structure for developing an application" [24]; Or "an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software" [25].

As discussed under the prerequisite of open IoT application architecture, we broadly define an "open IoT development framework" as **a non-proprietary, structured paradigm that can be used by any IoT application developers to implement the standard architecture of IoT applications**. It works as a publicly-known guideline that indicates what kind of components within an IoT application can be built and how they would interrelate; specify development interfaces, and sometimes offer development tools for using the framework. We argued that a comprehensive framework is supposed to abstract and isolate the developer from the complexity of the hardware and the networking sub-systems, re-define the development and re-usability of integrated hardware and software solutions. Thus developers are allowed to concentrate on the task logic itself.

We will briefly go through existing open IoT development frameworks that adopted different methods from this research, and explained why they were ruled out from our solution. And after introducing web services and their composition technologies, we will give a more detailed review on current solutions that similarly

anchored on IoT service composition, from which we picked a few representative ones as our competitors in following evaluations.

2.2.1 Process Virtual Machine Frameworks

Eclipse Kura is a Java/OSGi-based framework for IoT gateways. Its APIs offer access to the underlying hardware (serial ports, GPS, watchdog, GPIOs, I2C, etc.), management of network configurations, communication with M2M/IoT Integration Platforms, and gateway management. Java is a "write once, run anywhere" programming language and open source development platform that was originally aimed at set-top boxes, one of the first domains for non-desktop computing. Since 1995, JAVA has established its leading position in network application development because of cross-platform features. Today, Java Virtual Machine (JVM) can be running on most mainstream devices, from cloud server to smartphone, or even embedded microcontroller (by using JAVA SE Embedded). Hence, its feasible to provide a full stack distributed system based on JVM enabled devices.

By using JAVA implemented TCP/IP socket API, remote procedures running on heterogeneous devices, e.g. sensor/actuator integrated devices, edge routers and cloud servers, can work collaboratively as a distributed device network. JAVA also provides service interfaces for application development, such as RMI or Cobra, to make remote procedures more interoperable and reusable. Currently, several JAVA based IoT stacks have already been proposed. JVM provides more low level functions like direct hardware manipulation, and quicker remote access.

Process virtual machine is a mature technology in traditional computing network to achieve platform-independent interoperability by interpreting specific intermediate languages to hardware-dependent machine codes. Despite JVM, there are also a few notable counterparts in IoT area like virtual machine based on Python, i.e. MicroPython[1], .Net [8] and Java etc. However, PVM-based frameworks are usually language dependent and requires more computing resources to support VM, and also harder to integrate with other counterpart technologies. Another issue is that their development interfaces are mostly based on sheer programming, which entails technical expertise and may lead to a steep learning curve especially for beginners.

1 `https://micropython.org/`

2.2.2 Domestic Service Hub Frameworks

In home appliances areas, The home audio/video interoperability (HAVi) architecture is among the early home automation development frameworks. It consisted of a set of application programming interfaces (APIs), services, and an on-the-wire protocol specified by an industry initiative, which facilitates multi-vendor interoperability between consumer electronics devices and computing devices and simplifies the development of distributed applications on home networks [26]. It is based on the physical and link layers of the IEEE 1394 standard [27] and adopted the function control protocol and isochronous connection management protocol specified by IEC 61883.1. A HAVi implementation is a typical bus-structured home area network. A key feature of HAVi is that each physical device has an associated software proxy called device control module (DCM), which aggregates smaller units called functional component modules (FCMs) that allow application control of related device-function groups. HAVi APIs support both Java and Interface Definition Language (IDL), while the latter is a C-like representation and can map to different programming languages. Thus HAVi claims to be platform and language neutral. However, IEEE 1394 has gradually withdrew from smart device market in favor of new standards with high data speed such as 802.11ac.

Another example is OpenHAB [28]. OpenHAB played as the device hub for home automation, which was based on OSGi, a Java service-oriented development architecture that featured by modularity and runtime dynamics. It distributed domestic "add-on"s that mostly predefined bindings with physical hardware, external systems and web services. Supported home automation protocols and products included Z-Wave, Zigbee, MQTT, Chromecast etc. It provides users with a rule scripting method to define automatic behaviors of domestic devices, e.g. lighting, HVAC, security systems, water valves, IP video cameras etc. Users can customize and download necessary add-ons to local hubs. Due to that configuration and application logic are stored locally, it well protects users' privacy. Meanwhile it gets difficult to allow devices that belongs to different device networks to interoperate with each other.

Most of these frameworks leverage domestic service(e.g. Java service), which hinders cross-platform interoperability and service reusability by other systems. And though it is allowed to remotely control home appliances over Internet by connecting domestic appliance networks to some gateway devices. However, due to the particularity of home automation, they usually adopted a conservative, security-emphasized technical architecture that are not really devised for large-

scale, cross-domain deployment.

2.3 Mainstream Web Service Architecture

The entire web technology stack derives from a very simple vision to decouple network based systems into reusable, interoperable, composable data (web of data) and service (web of service) that any people (web for all) and any device (web on everything) can use it to share information [29]. This attempt, which makes web technology stack different from the traditional platform-dependent one, begins with decoupling and encapsulating data on the internet into generic resource. Though the definition of resource is gradually evolving from initially a document to almost any data source that can be uniquely identified by a URI nowadays [30]. To make resources addressable, accessible by any device and any people on the internet, a set of criteria to wrap up heterogeneous resources and provide a generic exchangeable form is necessary, such as: data schema, format, encoding etc, which is concluded as resource representation. Ideally, web resource representation is supposed to be both human and machine readable, and stay independently from lower layer of platforms and hardware, as well as the upper services and applications who use the resource. It is a well-exploited research area with topics like metadata description (xml, json, exif etc.), data schema (DTD, XSD, MODS etc.), resource description framework (RDF, json-LD, microdata etc.), web ontology (OWL, SSN etc). IoT resources need alike representations, but with far more complicated properties and behavior patterns to be modeled and described than traditional ones.

The next effort of web research community is to decouple computing system functionality into uniformly accessible components who consumes the resources to provide services for both machine and human users. This effort resulted in a few full-fledged distributed computing methods, namely web service, examples like SOAP, REST, XML-RPC. Compared with their platform-dependent or language-dependent ancestors, including: RPC (Remote Procedure Call), RMI (Remote Method Invocation, examples like CORBA, EJB), web services are believed to be self-contained, platform-independent, and reusable modules that provide standard functionality. They can be published, discovered, located, invoked, and loosely coupled throughout the web, and facilitate the integration of newly-built as well as legacy applications both within and across organizational boundaries [31]. These advantages make web service more scalable and a better choice for internet scale

applications.

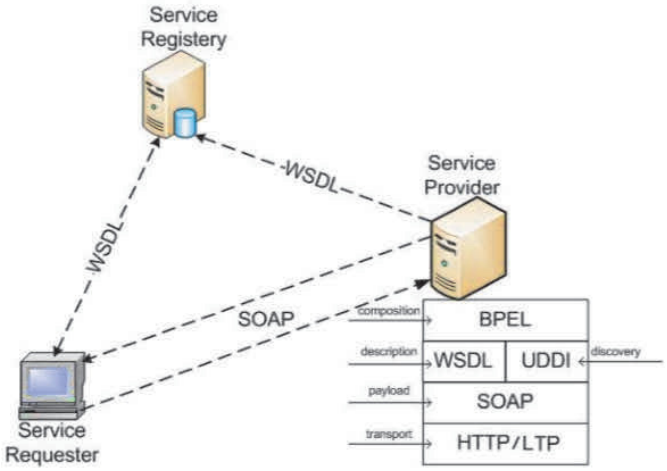
The W3C consortium defines a Web service as "a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards". IBM defines Web services as "self-describing, self-contained, modular applications that can be mixed and matched with other Web services to create innovative products, processes, and value chains. Web services are Internet applications that fulfill a specific task or a set of tasks that work with many other web services in a manner to carry out their part of a complex work flow or a business transaction" . According to Microsoft, "A Web Service is a unit of application logic providing data and services to other applications. Applications access Web Services via ubiquitous Web protocols and data formats, such as HTTP, XML, and SOAP, with no need to worry about how each Web Service is implemented" . HP defines Web services as "modular and reusable software components that are created by wrapping a business application inside a Web service interface. Web services communicate directly with other web services via standards-based technologies" . SUN perceives a Web service as an "application functionality made available on the World Wide Web. A Web service consists of a network-accessible service, plus a formal description of how to connect to and use the service" .

As we are able to integrate different smart things with various capabilities into the Web, the next logical step we shall consider is how to abstract those devices into reusable web services other than simple static or dynamic web pages. Conventional wisdom has it that there are two major paradigms of web services: REST-compliant Web services and arbitrary Web services. The primary purpose of the service is to manipulate web resources using a uniform set of "stateless" operations in the former one while using an arbitrary set of operations in the latter one. Both paradigms can be adopted by smart things or smart gateways.

2.3.1 WS-* Architecture

It is usually referred as WS-* for Web Services that use Simple Object Access Protocol (SOAP) messages with an Extensible Markup Language (XML) payload and a HTTP-based transport protocol to provide remote procedure-calls (RPCs) between clients and servers. It has been popular in traditional enterprises and

widely used in enterprise machine-to-machine (M2M) systems. The key technologies of WS-* are SOAP, Web Service Description Language (WSDL), Universal Description Discovery and Integration (UDDI) and Business Process Execution Language (BPEL) as shown in Figure 2.2.



Source: Deze Zeng et al [32]

Figure 2.2: WS-* Workflow and Protocol Stack

LTP [33] is a light-weight Web service transport candidate protocol that allows transparent end-to-end exchange of Web service messages between resource-constraint devices and server or PC class systems. LPT's structure resembles that of WS-messaging and utilizes transport binding and compressed SOAP that is fully compliant to SOAP standard. The main features of LTP are platform independence, low resource consumption and implementation-agnostic definition of the protocol.

SOAP [34] is an XML-based protocol to let applications exchange information over HTTP. A SOAP interface is typically designed with a single URL that implements several RPC methods, which define a message architecture and format, hence providing a rudimentary processing protocol. The top-level XML element of SOAP message is called envelop, which includes two XML elements: header and body. The header specifies routing and Quality of Service (QoS) configuration while the body contains the payload of the message indicating the interoperations.

WSDL [35] is an XML-based language describing Web services as a collection of communication end points that can exchange messages. In other words, a WSDL document describes a Web services interface and provides users with a

point of contact. The SOAP messages and sequences are abstractly described by WSDL. A WSDL port type contains an abstract set of operations supported by endpoints. The WSDL binding links the set of abstract operations with concrete protocol and data format specification for a particular port type. WSDL describes service interface, which are independent of the service implementation endpoint and how the services are implemented.

UDDI [36] is a platform-independent, XML-based registry framework for describing and discovering worldwide Web services. It can be viewed as a directory of WSDL-described web services. Web services can be registered and located in the directory. It can be requested using SOAP messages to provide access to WSDL documents, which describe the protocol bindings and message formats required to interact with the web services listed in its directory.

BPEL [37] defines a notation for specifying process behavior based on interactions of Web services. Web service interactions can be described in two ways: executable processes and abstract processes. Both can be modeled by BPEL. Executable processes model actual behavior of a participant as interactions while abstract processes describe observable behavior and/or process template. BPEL extends the WS-* interaction model to enable business transactions. BPEL defines an interoperable composition model that enable the extension of automated process integration both within and between businesses.

One may first notice that HTTP performs as transport protocol at the lowest level. Above that, SOAP handles the interaction between services. WSDL and UDDI concern the description and discovery of services at the next higher level. BPEL actually deals with the composition of services at the highest level. Now we look at how these technologies work in a WS-* workflow. Suppose all the available services have registered in the Service Registry. Service Requestor sends a service lookup request described by WSDL to Service Registry. If a suitable candidate service is found, its description is returned to the Service Requestor. Then Service Requester and Service Provider establish connectivity and communicate with each other using SOAP according to the description.

The use of WS-* for smart things dates back many years ago. A Service-Oriented Device Architecture (SODA) [38] is proposed to integrate a wide range of physical devices into distributed IT enterprise systems. In SODA, all the sensors and actuators are exposed as abstract business Web services to the programmers. A bus adapter locates in the boundary between the cyber world and physical world realms, and talks to proprietary and standard device interfaces

but presents an uniform Service-Oriented Architecture (SOA) services. Pintus et al. [39] also proposed a SOA framework where smart things were described using WSDL standard and logical connections between smart things are modeled as web services orchestrations using the BPEL language. The SOA approach for networks with embedded systems can be also found from many other projects, such as SIRENA [40] and SOCRADES [41].

2.3.2 REpresentational State Transfer

The term REST was first coined by Roy Fileding in his PhD thesis [42], which is considered as the "true architecture of the Web". The basic concept of REST is that everything is modeled "resource", or particularly HTTP resources, with a Universal Resource Identifier (URI). The REST architectural style is based on the following four principles [43]:

1. Resource identification through URI. All the resources exposed by RESTful web services are identified by URIs. Through URI, the clients can identify their interaction targets. A global addressing space is provided for service and resource discovery.

2. Uniform interface. RESTful services treat the HTTP as an application protocol instead of a transport protocol in WS-*. Therefore, the term REST is often used in conjunction with HTTP and the RESTful resources can be manipulated using HTTP verbs such as PUT, GET, POST and DELETE. PUT creates a new resource while DELETE deletes it. GET retrieves the current state of a resource in some representation while POST updates a resource with new state.

3. Self-descriptive messages. Resources are decoupled from their representations such that it is free to use a variety of data formats to describe themselves provided that the appropriate representation formats are agreed and understandable by endpoints. For example, the data can be in any common-used formats such as HTML, XML, plain text, PDF, and JPEG. Metadata about the resource can be used to control caching, detect transmission errors, negotiate the representation format, and perform authentication or access control between endpoints.

Notice that although ReST is initially described in the context of HTTP, it is not limited to that protocol. RESTful architectures can be based on any other

application layer protocols if they can provide a rich and uniform vocabulary for applications to transfer meaningful representational states. By this way, the potential of existing well-defined network protocols can be reexploited without additional efforts.

IoT services designed in accordance with the ReST paradigm exhibit very strong similarity with traditional web services, thus greatly facilitating the adoption and use of IoT by both end users and service developers, which will be able to easily reuse much of the knowledge gained from traditional web technologies in the development of services for networks containing smart objects [44]. In ReST, a service is defined by a set of states (similar to variable in OOP but not identical since a state may contain several variables) and state transfers (similar to function or method in OOP but not identical). A service in client can remotely invoke the state transfer of a service in server by sending a desired state transfer message over HTTP. Furthermore, this HTTP message should contain all the necessary information to accomplish the state transfer, which means each ReST service in server will not store any state (for example: user name, access token, or resource lock) from any other service for future use, i.e. stateless server. This feature further simplified the composability: if a service is composable from the beginning, it will not become non-composable caused by sequential problems, such as: deadlock, because each state transfer is independent.

To our best knowledge, the RESTful architecture is preferred for IoT mainly for its two features. One is its low complexity and the other is its loose-coupling stateless interactions. The two features enable web servers in the RESTful architecture to be embedded into resource constrained devices (e.g. Resource-oriented architecture) and also enable easy composition of web services. For example, REST can be the architecture of choice for tactical, ad hoc integration over the Web (i.e., mashup) [43]. The previous work on integrating sensor networks to the Internet showed that the lightweight aspect of REST made it an ideal candidate for resource-constrained embedded devices to offer services to the world [45, 46]. To support this opinion, the feasibility of using RESTful web services was demonstrated with an evaluation of performance and power consumption in an IP-based multi-hop low-power sensor network [47].

2.3.3 Operation-based Paradigm v.s. State-based Paradigm

The decoupling efforts on web has been carried out for many years, various methods have been put forward. As we summarized previously, current researches on decoupling web system into web resource and web service has already established a solid foundation for composability. And among the aforementioned service architectures, XML-RPC and SOAP can be concluded as the representative technology for operation based paradigm, while REST belongs to the state-based paradigm.

The naming of the two paradigms comes mainly from their difference in providing access interface (though there also exists many differences in their implementation details accordingly): the operation based services can be remotely invoked by sending a desired operation message (for example: $< operation > openlight < /operation >$) to service specific URI (`www.domain.com/lightservice`) over HTTP POST request, while resource based services can be invoked as the same manner as a resource, i.e. using one of the HTTP request (POST to CREATE resource, GET to READ resource, PUT to UPDATE resource and DELETE to DELETE resource) together with desired state transfer ($< state > on < /state >$) to the target resource URI (`www.domain.com/light/onoff`). Currently, the state based methods are better choices for composable system due to some critical composability features: **loose-coupling**, **generic interface** and **statelessness**.

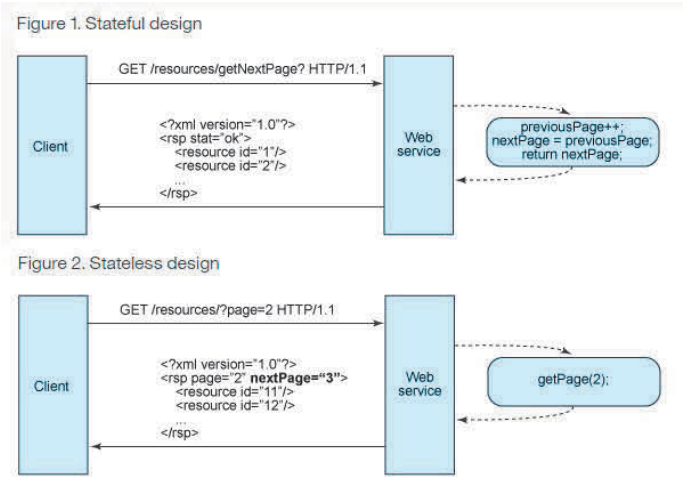
First, loose-coupling is the major target for decoupling efforts on web. Resource based service is widely accepted for its outstanding loose-coupling and light-weighted features to provide services across organizational boundary.

Second, the generic interface to access both resource and service will help to simplify requirements of the uniform messaging mechanism between all web components. Currently, ReST is the only architectural style that expose its access interface as standard HTTP operator (GET/PUSH/PUT/DELETE) and message (XML, JSON or other web standard). In RESTful architecture, the service can be taken as a special form of resource since they share the same access interface, i.e. the HTTP operation. The establishment of this uniform access interface simplified the complex web resource and service composability into a relatively easier goal: the composability of web resources via HTTP operation.

Third, statelessness will help to maintain composability for web system. It requires Web service clients to send complete, independent requests; that is, to send requests that include all data needed to be fulfilled so that the components in the intermediary servers may forward, route, and load-balance without any state being

held locally in between requests. It does not require the server, while processing the request, to retrieve any kind of application context or state. Statelessness improves Web service performance and simplifies the design and implementation of server-side components because the absence of state on the server removes the need to synchronize session data with an external application.

Figure 2.3 explains the differences between a stateful service and a stateless service, where a stateful service from which an application may request the next page in a multipage result set, assuming that the service keeps track of where the application leaves off while navigating the set. A stateless service, on the other hand, in the request for a multipage result set, the client should include the actual page number to retrieve instead of simply asking for next.



Source: V Dambal [48]

Figure 2.3: Stateful v.s. Stateless

Hence, by applying state based service architecture, such as ReST, a composable web system is defined by a set composable resources, and the composable resource is defined as: first, it wraps data (with state) and computing service (with state transfer) as web accessible component which can be interacted with standard HTTP operator and message; Second, it can exchange state information and trigger desired state transfer accordingly by sending and receiving generic state descriptions (usually called resource representation which contains data, metadata and sometimes metadata of metadata to describe states) carried by HTTP message.

Contrary to state based paradigm, operation based paradigm is conceptually like functional programming: a service exposes its functionality (for example:

make a sound) as a operation (like a function in functional programming) that can be invoked by other services. The invoking mechanism can be implemented by direct invoking (using operation name, parameters and returned values), event based invoking (using event to trigger operation and setting specific inner states, this may break the service encapsulation to achieve more control possibility, for example: one event for start the sound, another one to cease it). While the state based paradigm is a little like OOP without public methods: a service is defined by a set of states (similar to a set of member variables in OOP) and state transfers (used to change states, similar to methods in OOP). However, no state transfer can be directly invoked. it can only be triggered by messaging a desired state (for example: if a light service is at its state of "off", messaging a desired state "on" may trigger a state transfer to turn on the light. However, it cant be guaranteed since there may exist no available state transfer to actually achieve desired state from the current state). Two messaging mechanism is usually used to deliver desired states: plain HTTP message, or specifically formatted message (for example XML) over HTTP.

The two paradigms have their own advantages and disadvantages, which should be considered thoroughly according to requirements and constraints. Generally, operation based implementation is easier to be programmed and performed because of its functional nature, while state based implementation is more loosely coupled and thus can better support reusability and composability.

2.4 Web Service Composition

Composability is defined as "the ability to agilely create, configure and test a unique system by selecting and assembling models/modules from a pool of reusable components in various combinations to satisfy specific user requirements" [49]. Similarly, Davis et al. defined composability as "capability to select and assemble components in various combinations to satisfy specific user requirements meaningfully" [50]. The prerequisites of composability comprise of modularity and interoperability [51]. Particularly within web context, modularity requires to equally encapsulate each and every component into web-accessible service that will be used as the basic block for building entire systems. While interoperability emphasizes that all blocks must be exposed in standardized interfaces to enable unified butt-to-butt joint among different blocks. If we compare a block to a puzzle piece, its interface is as important as the concave and convex part of each

puzzle piece, and is basically described in a state-based way in this research.

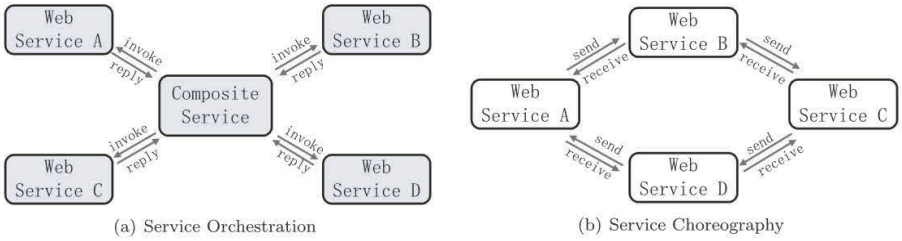
In web application domain, web service composition can be achieved by three kinds of engineering methods: **direct invoking**, **event based invoking** and **messaging**. Direct invoking method remotely calls a procedure using its name and arguments, as if it's a local function. It is usually language dependent, e.g. SWORD [52], Ericsson's JAVA based IMS composition system [53]. Event based invoking method uses events to invoke remote services asynchronously, e.g. HP's eFlow [54]. Messaging method usually relies on Web messages to trigger remote services, which can be further divided into plain HTTP message and formatted message (for example XML) over HTTP, e.g. JOpera [55], SABRE [56], Yahoo Pipes [57].

As we have discussed in previous subsectionn, in comparison with operation-based interfaces, state-based interfaces are supposed to let users achieve desired system state by specifying related endpoints instead of invoking internal operations and mechanism. Thus, it presses on towards a black-box model and contributes to simplifying composition procedure as well as maintaining a stateless design style in real practise.

2.4.1 Service Composition

In a composable web service system, service composition is defined as the process of combining different Web services to provide a value-added service [58]. Web service composition is different from traditional application integration, where applications are tightly coupled and physically combined. Web services adopt a document-based messaging model, which supports the integration of loosely coupled applications that are across multiple organizations. Service composition is becoming the most promising way to integrate cross-organizational applications on the Web, especially in enterprise and consumer domain [58–60]. There are two ways to describe the sequence of activities that make up a business process: **orchestration** and **choreography** [61], as shown in Figure 2.4.

Orchestration represents a single executable process that coordinates the interaction among the different components, by describing a flow from the perspective and under the control of a single endpoint. It can therefore be considered as a construct between an automated process and the individual components that enact the steps in the process. Service Orchestration has been widely deployed in business platforms, e.g. IBM Business Process Manager, Oracle BPEL Process Manager [62]. The orchestration is usually defined by BPEL or BPML to



Source: C Peltz [61]

Figure 2.4: Service Orchestration v.s. Service Choreography

compose various services implemented by COBRA, SOAP or REST [63]. Besides business solutions, there also exist open source execution engines, e.g. Apache ODE, ActiveBPEL [64].

Different from Orchestration that always represents control from one party's perspective, Choreography is more collaborative and allows each involved party to describe its part in the interaction. Choreography represents a global description of the observable behavior of each of the services participating in the interaction, which is defined by public exchange of messages, rules of interaction and agreements between two or more endpoints. It is typically associated with the interactions that occur between multiple web components rather than a specific process that a single party executes, and are particularly useful in those situations in which multiple parties have to collaborate, however are not executable, and must be implemented inside of each component individually. Most of its applications were research prototypes, e.g. Let's Dance [65], or proposed to improve WS-CDL standard, examples like [66–68].

2.4.2 Web Service Composition Category

Current web service composition under orchestration can be divided into three categories: **process/programming based composition**, **interaction-based composition**, and **planning-based composition** [69]. Most existing Web service composition techniques require programming to some extent for constructing the orchestration model [54, 70, 71]. Composers first need to study the component services that are described using WSDL or some ontology languages, and understand the functionalities of the services and the supported operations. A further step analysis requires to identify the way operations are interconnected, services are invoked, and messages are mapped to one another. The process-based compo-

sition scheme makes the process of composing service demanding for composers. Composers need to be domain experts who are familiar with the service description language, the service orchestration algebra, and the corresponding programming skills. Since common users cannot act as a service composer, the programming based scheme hinders common users from composing Web services at large.

The interactive composition scheme blurs the distinction between composers and common users. Composers are required to have a clear goal and know the tasks that need to be performed to accomplish the composition. Common users can be guided through a set of steps to finish a composer's task. The composition scheme will work interactively with the common users to help them achieve the orchestration model. The orchestration process can start from users' goals and work backward by chaining all related services. It can also start from some initial states and achieve the users' goals by adding services in the forward direction. At each step, the scheme will choose a new service based on the task specified by the users. The interactive scheme can also capture the constraints and preferences during the interaction process. The constraints and preferences can serve as additional criteria to select services for the composition.

The planning-based composition scheme aims to relieve users from the composition processes as much as possible. It relies on AI planning techniques for automatic service composition. In this context, users are allowed to submit a declarative query specifying the goal he/she wants the composite service to achieve together with some of the constraints and preferences that need to be satisfied. Based on the user's query, the composition scheme can derive a corresponding orchestration model with all constraints and preferences satisfied. The planning scheme regards services as actions that are applicable in states. State transitions are specified using the preconditions of some actions. A transition will lead to some new states, in which the effects of some actions are valid. Based on this, the composition scheme recursively adds new services until users' goals have been achieved. The states of existing service in the orchestration will determine the selection of the new services. For example, the preconditions of the new services should be satisfied via the effect of some existing services.

2.4.3 Web Service Composition Approach

When it comes to real practise, there are a few approaches to substantiate a web service composition system [72].

BPEL. BPEL is an XML language that supports process oriented service composition. Developed by BEA, IBM, Microsoft, SAP, and Siebel, BPEL is currently being standardized by the Organization for the Advancement of Structured Information Standards (OASIS) [2]. (Sun Microsystems recently joined the OASIS technical committee as well.) BPEL composition interacts with a Web services' subset to achieve a given task. In BPEL, the composition result is called a process, participating services are partners, and message exchange or intermediate result transformation is called an activity. A process thus consists of a set of activities. A process interacts with external partner services through a WSDL interface.

Semantic Web (OWL-S). The Semantic Web vision is to make Web resources accessible by content as well as by keywords. Web services play an important role in this: Users and software agents should be able to discover, compose, and invoke content using complex services. The DARPA Agent Markup Language (DAML) extends XML and the Resource Description Framework (RDF) to provide a set of constructs for creating machine-readable ontologies and markup information. The DAML program's Semantic Web contribution is the Web Ontology Language for Services. OWL-S (previously known as DAML-S) is a services ontology that enables automatic service discovery, invocation, composition, interoperation, and execution monitoring.

Web Components. The Web component approach treats services as components in order to support basic software development principles such as reuse, specialization, and extension. The main idea is to encapsulate composite-logic information inside a class definition, which represents a Web component. A Web component's public interface can then be published and used for discovery and reuse.

Algebraic Process Composition. Algebraic service composition aims to introduce much simpler descriptions than other approaches, and to model services as mobile processes to ensure verification of properties such as safety, liveness (correct termination, for example), and resource management. Mobile-processes theory is based on π-calculus, in which the basic entity is a process-it can be an empty process; a choice between several I/O operations and their continuations; a parallel composition; a recursive definition; or a recursive invocation. I/O operations can be input (receive) or output (send). For example, $x(y)$ denotes receiving tuple y on channel x; $\bar{x}[y]$ denotes sending tuple y on channel x. Dotted nota-

2 `www.oasis-open.org`

tion specifies an action sequence, such as $\bar{c}[1, d].d(x, y, z).\bar{c}[x + y + z]$, in which a process sends tuple $[1, d]$ on channel c, then receives a tuple at channel d whose components are bound to the variables x, y, and z, and finally sends the sum of $x + y + z$ to channel c. Parallel process composition is denoted with $A|B$. Several processes can execute in parallel and communicate using compatible channels.

Petri Nets. Petri nets are a well-established process-modeling approach. A Petri net is a directed, connected, and bipartite graph in which nodes represent places and transitions, and tokens occupy places. When there is at least one token in every place connected to a transition, that transition is enabled. An enabled transition might fire by removing one token from every input place, and depositing one token in each output place. We can model services as Petri nets by assigning transitions to methods and places to states. Each service has an associated Petri net that describes service behavior and has two ports: one input place and one output place. At any given time, a service can be in one of the following states: not instantiated, ready, running, suspended, or completed. After we define a net for each service, composition operators perform composition: sequence, alternative (choice), unordered sequence, iteration, parallel with communication, discriminator, selection, and refinement. These operators guarantee the closure property. Thus, by composing two or more Web services, we produce another service.

Model Checking and Finite-State Machines. Other approaches for Web service composition include model checking, which aims at modeling service composition as Mealy machines, and automatic composition of finite-state machines (FSMs). Model checking is used to formally verify finite-state concurrent systems. We describe system specification using temporal logic, then traverse and check the model to see whether the specification holds. We can apply model checking to Web service composition by verifying correctness inside a workflow specification. Among the properties we can check are data consistency, unsafe state avoidance (deadlock), and business-constraint satisfaction.

2.5 IoT Service Composition: Parallel Research

Service composition is an emerging research genre that widely thought as a key method to quickly deliver new functionalities to IoT applications [73], improve re-usability, lower down development cost [74], and further attract open participation [75]. As service composition for IoT applications is an emerging research

genre within recent decade and still in its nascent stage, there is no matured enterprise solutions or business products up until now to the best of our knowledge. However, it has drawn the interests of open source communities. In this section, a few fresh open source projects and research prototypes were selected as the representative parallel researches to examine the similarities as well as differences regarding research methodology and approaches.

2.5.1 Programming/Process-based Composition

Programming/process based composition is the most widely used method which constructs business logic of service composition by manual programming mostly. This kind of methods can be further divided into two sub-groups: traditional program language based, and domain specific language based. Traditional language based methods can offer users the most powerful toolset to define complex logic and support a wide range of tasks. Examples are: PubNub, OpenIoT. However, it is technically demanding and not optimized for composition tasks. Hence, many composition methods prefer to defining their own language based on domain knowledge, e.g. domain specific language, to lower down the learning curve and better suit composition requirements. Examples are Sensorpedia, SM4RCD.

Figure 2.5: an example of editing PubNub process

Though not exclusively centering on IoT application development, PubNub is a realtime publish/subscribe messaging API built on a global data stream network.

It provides realtime infrastructure-as-a-service for data streaming and device signaling which allows user to establish and maintain persistent connections based on WebSockets, Socket.IO, SignalR, WebRTC data channel and other streaming protocols. Based on PubNub, Eon is an web IoT visualization development kit that widely used to provide live sensor data dashboard composition with platform-agnostic messaging and realtime data steaming. While the development requires for sheer manual JavaScript programming and hence results in a steep learning curve and high kick-start barrier.

Sensorpedia [76] aimed to organize and provide access to online sensor network data following social media principles, the development of which was divided into a web-based applications and the supporting web services interface. The former provided a map-based mashup interface for browsing and discovering available sensor data by location and keywords, while the latter offered an Atom-model-based application programming interface and supported multiple data representation including GeoRSS, SensorML etc.

Nils G et al in [77] proposed a model driven development paradigm based on a domain specific language for describing states and state transitions of state machines, named State Machine for Resource Constrained Devices (SM4RCD). The major difference between SM4RCD and this research lied in that SM4RCD specified a SOAP message compression other than RESTful architecture. It also utilized Lean Transport Protocol (LTP), and further generated C++ code towards target systems.

OpenIoT project [78] was first known to public in 2012 and co-funded by the European Commission. It aimed to provide "a middleware platform enabling the semantic unification of diverse IoT application in the cloud". OpenIoT platform adopted Extended Global Sensor Networks (X-GSN) [79] to collect, filter and combine data streams from virtual and/or physical sensors. Specifically for mobile sensors, A Cloud-based Publish/Subscribe middleware (CUPUS) was leveraged. A Linked Stream Middleware(LSM) acted as a cloud storage for storing both/ data streams and metadata, which supported extended W3C SSN ontology and SPARQL queries. Recently, OpenIoT is also researching on flow-based composition method. However, there's litter information about the new method.

2.5.2 Rule-based Composition

Rule based composition may be the oldest method that used in many circumstances. rules are usually pre-defined and represented as events, conditions, for-

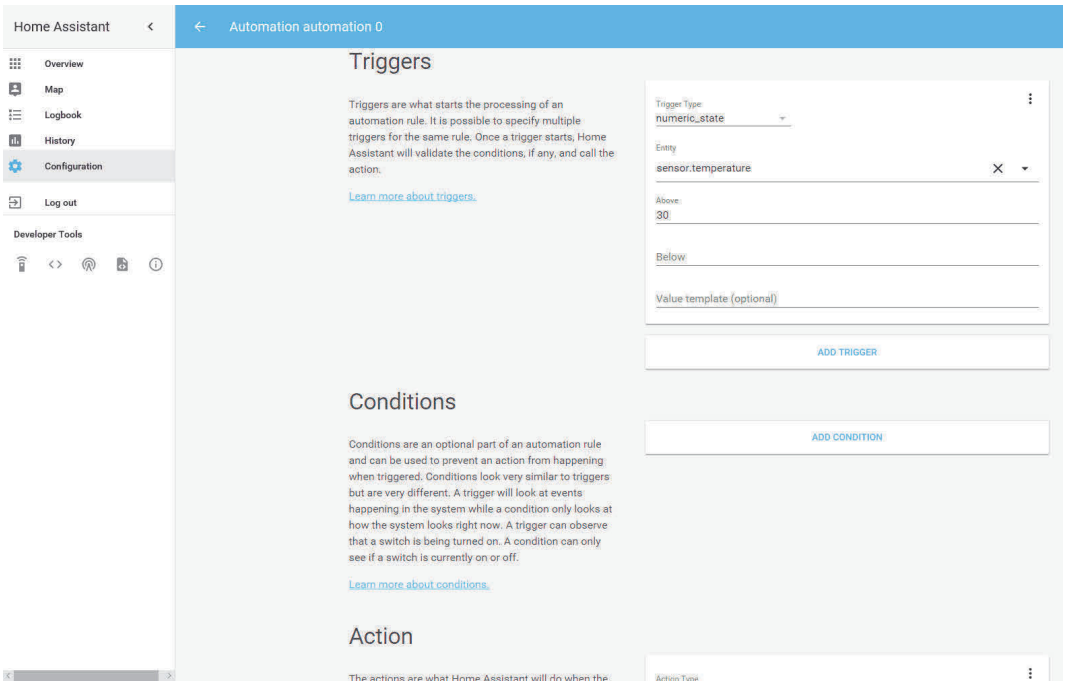


Figure 2.6: an example of setting Home Assitant rule

mulas, or symbolic logic. Whenever a rule is met in run-time, the corresponding operation will be triggered automatically.

Home Assistant is a typical rule-based home automation hub that running on Python, which enables event-triggered device observation, control and automation. It predefined a set of standard entities, related properties as well as functions. Developers must wrap up IoT devices and expose their interfaces according to the stipulation. The advantages lie in the bidirectional synchronization between the states of physical and virtual entities and easy to obtain automatability. However, it entails re-adpation to introduce existing services and language-dependency also increases system couplingness and the difficulty for components to be reused.

2.5.3 Flow-based Composition

Flow based composition is very popular in recent years, because it can easily be represented in graphic UI, for example as a directed graph of interconnected nodes. Hence, the graphic UI is usually more intuitive and easy to use. However, when it comes to complex task logic that hard or even impossible to be defined in

flow UI, it may still introduce programming language into the composition. One typical example is NodeRed[3].

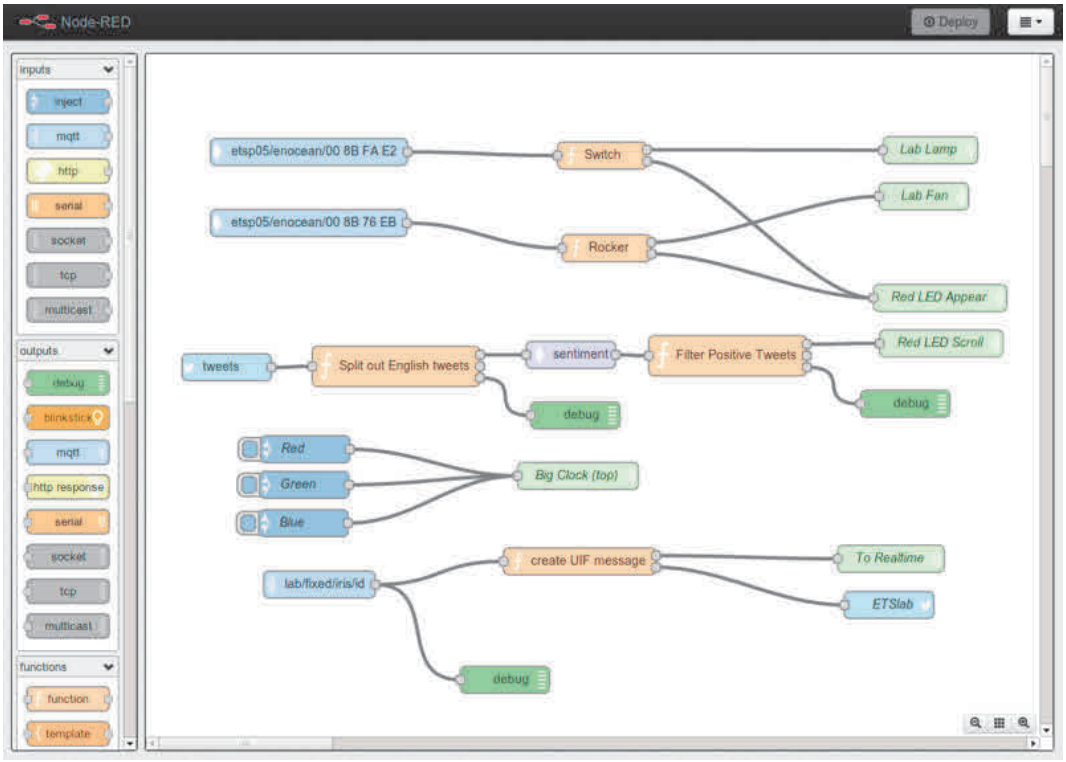


Figure 2.7: an example of defining Node-Red flow

Node-Red is an open flow-based IoT development tool initiated by IBM's Emerging Technology Services in 2003. It provides a mash-up style editor that allows users to drag-and-drop different widgets, including IoT devices, APIs, web services etc., and wire them together. Node-Red is based on node.js and uses an event-driven, non-blocking I/O model to create applications that run across distributed devices. This intuitive interface with rich visual elements has attracted a large group of users within IoT communities. However, as each type of node is described as an opaque widget and behave very differently from one another, it costs user extra learning effort each time when a new type of node comes in the flow. And when it comes cross-domain calls, there will also be a difficult issue that how to obtain necessary information in order to correctly configure a node.

3 `https://nodered.org`

SensorMasher [80] adopted a semantic approach of linked data to manage sensor mashups. Users were able to visually drag and drop sensors as data sources and connect them to data processing blocks to create composite data sources. Multiple data formats including RDF, JSON, XML, RSS etc. and SPARQL based query were supported.

WoTKit [81] served as a data aggregator, visualization, remote control and processing tool for Web of Things, which was based on Java web application and Spring Framework. It also had a RESTful API that supported CSV, KML, HTML and JSON formatted data and graphic-element-based user interface.

Vital-OS is a IoT-smart city research project funded by European Union's Seventh Framework Programme, which is dedicated to enable the integration and semantic interoperability of multiple IoT systems, while adding complex data processing and tools to easily build applications exploiting all the underlying data and services [82]. To achieve this goal, Vital-OS designed a service-oriented architecture comprised of 3 layers: (IoT and data) Resource Access Interface, Service, and Service Access Interface. A set of tools and components were included: Governance and Monitoring Toolkit, Data Management Service, IoT Adapter, IoT Service Discovery, Development and Deployment Toolkit, Orchestrator, Data Management Service, ICOs and Services Discovery, Complex Event Processing, Filtering Service etc [83, 84]. Each toolkit or component in Vital-OS was replaceable, and communicated with each other using events, which made it a loose-coupled architecture.

A more detailed subjective comparison will be provided in Chapter 5.

2.6 Summary

In the beginning of this chapter, we provided a layer-structured overview of mainstream Internet of Things technology stack and defined "open IoT development framework" as "a non-proprietary, structured paradigm that can be used by any IoT application developers to implement the standard architecture of IoT applications.". We argued that a comprehensive framework is supposed to abstract and isolate the developer from the complexity of the hardware and the networking subsystems, redefine the development and reusability of integrated hardware and software solutions.

Followed by a systematical review on existing open IoT development frameworks. Grouped by respective core technology used, current mainstream open IoT

frameworks can be roughly separated into three major genres:1) Process Virtual Machine based frameworks; 2) Domestic Service Hub based frameworks; And 3) IoT Service Composition based frameworks. In section 2.2, we analyzed the underlying mechanism of the first two genres, which reflected some drawbacks at: 1) High customization cost; 2) High expertise requirement and kick-start barriers; 3) Low component resuability and 4) Difficulties in deploying geographically dispersed, cross-domain scenarios. And that's why we turned to the last genre, IoT service composition in this research.

To understand IoT service composition, detailed explanations were made to introduce Web Service technology stack in section 2.3 and service composition in section 2.4. The main branches of Web service architecture included operation-based WS-* paradigm and state-based RESTful architecture, while the latter was preferred by this research due to its features, e.g. loose-coupling, generic interface and statelessness, better service composability.

Followed by the introduction of web service composition and its categories. Firstly, we defined "composability" as "the ability to agilely create, configure and test a unique system by selecting and assembling models/modules from a pool of reusable components in various combinations to satisfy specific user requirements". Service composition also had two variations: service composition and service choreography. As most service choreography application were still at research prototype stage, the terminology "service composition" in this research, if not specifically noted, all refer to "service compostion".

In section 2.5, we focused on IoT service composition explicitly. Existing IoT service composition based frameworks could be divided into: 1) Programming/ Process-based composition, 2) Rule-based composition, and 3) Flow-based composition. We selected one representative example from each category as our parallel researches to be compared in the evaluation.

Chapter 3
Approach

3.1 Research History

Before coming to the topic, firstly, we want to survey previous literature and listed a few that possessed some kind of theoretical continuum with this research. Though not all of them are necessarily related to the IoT field, these ancestor technologies and researches have shared a very similar idea to provide a standard device description for heterogeneous devices and hardware modelling, which hopefully can help readers to understand the base of our solution model in this research.

Controller Area Network(CAN bus). Controller Area Network (CAN bus) was first developed at Robert Bosch in 1980s, and its extended version became an ISO standard in 1993 [85]. Originally, it was devised for enabling message-based communication among different subsystems built in an automobile, e.g. ABS, electric power steering, engine control unit etc., hence able to establish a feedback control among multiple sensors, actuators and micro-controllers. This kind of interconnections, which allowed value-added features to be implemented using software, avoided extra cost and complexity that may be caused by traditional hard wiring way [86].

In a CAN network, data was conveyed and transmitted via a uniquely identified message and no individual nodes were addressed. When a node wanted to transmit information, it needed to pass the data and the identifier to its CAN controller and set the relevant transmit request. It was the CAN controller that formatted the message contents and transmitted the data in the unified CAN frame. Once the node gained access to the bus, all other nodes became receivers and performed an acceptance test according to the message identifier, to determine whether the received data was relevant to particular device or not. This was known as the "producer/consumer" mechanism, whereby one node produced data on the bus for

other nodes to consume, and thus CAN bus was able to perform communications on peer-to-peer, multicast or broadcast basis. And it required no interaction from a bus master or arbiter.

CAN bus provided high-speed serial interface, low-cost physical medium as well as economical and prompt data communication. However, similar to other bus-structured device network, the amount of nodes that single bus can support was inherently restricted and communication performance significantly dropped as nodes increased, which made CAN bus an inappropriate solutions for large-scale, disperse IoT deployment.

MIB and SNMP. A management information base (MIB) was a virtual information store used for managing the entities in a communication network [87]. It was often associated with the Simple Network Management Protocol (SNMP) [88] to provide network manageability over TCP/IP implementations. Objects in the MIB were defined using a subset of Abstract Syntax Notation One (ASN.1) [89]. Each type of object had a name uniquely identified by an administratively assigned object identifier (OID), a syntax, and an encoding. The SNMP modeled all management agent functions as alterations or inspections of variables. Through the uniform interface provided by MIBs deployed on both the network management stations and the agents in the network elements, SNMP was able to recognize and communicate management information. Thus, a protocol entity on a logically remote host (possibly the network element itself) interacted with the management agent resident on the network element in order to retrieve (get) or alter (set) variables. The strategy implicit in the SNMP was that the monitoring of network state at any significant level of detail was accomplished primarily by polling for appropriate information on the part of the monitoring center(s). Though SNMP explicitly minimized the number and complexity of management functions, still resource constraint devices might not be able to equipped with management agents responsible for performing the network management functions requested by the network management stations. Also, network elements tended to be stable devices such as hosts, gateways, terminal servers etc., and it remained to be difficult to maintain and update the information stored in MIBs in realtime manner when it came to devices with high mobility and fluctuant network topology.

Line Printer Daemon. Line Printer Daemon managed the printer spool area and the print queues, which started at boot time of Linux and BSD systems by default. When LPD started, it read the /etc/printcap file to find out about

the printers available for its use. The printcap file defined the printers and their characteristics. If the printer itself support Socket API, remote access could be defined in printcap file for Socket connection directly to the printer. LPD relied on TCP/IP as the communication protocol and adopted a domain specific language to define service functionality. Since the printer itself had no resources for data storage, computation and network management, a UNIX/LINUX computer was usually required to provide necessary functionalities. There were several limitations about LPD: First, it was dependent on specific operating sources and Socket communication. Second, the LPD file was specifically devised for describing printers and their functionality, and lacked a uniform model to be extended to other sensors, actuators and IoT devices. Hence, there was no easy way to apply this domestic service technology to IoTs. The advantage, compared with web services, was the optimization which may lead to a better service quality in specific domain and local use, such as: lower latency, more service functionality etc.

3.2 State-based Composable Service Interface

Instead of obtaining data directly from physical sensors/actuators, wireless sensor network or an IoT device, the focus of this research lie on acquiring sensor/actuator data via standard web service interfaces, such as popular online sensor data platforms, embedded web servers within smart IoT devices etc., since exploiting web service as data input service node is also an important interest for our proposed paradigm.

However, not all web-accessible IoT resources can naturally be regarded as composable IoT services. The composability requirement of IoT resource entails a composition-oriented service interface model that not only provides consistency with ordinary web services, but also able to express particular physical properties and functionality innately derived from hardware devices. To legibly specify the prerequisites for composable IoT service interface within our proposed framework, we are expecting it to be:

1. An IoT resource and its host service must be addressable and accessable via a **unique resource identifier**, i.e. URI-deferencable, within a composition.

2. The exposed interface of host service must be based on **states** rather than operations. A state-based service “only stores states of a service, and thus

need to derive transitions by comparing previous state and its successor" [90].

Though the inner mechanism of each host service may differ from one another, we insist that the outer interface it exposes must be uniformly modelled into a set of "states", where any operation can be described by the transition of one or more states. Intuitively, the **state** of a system is its condition at a particular point in time [91]. In software engineering, a state is defined by a set of **variables** and specific **values** [92].

According to engineering definition, a set of states S is usually specified by a collection of variables V and their ranges R, which are sets of values. A state $s \in S$ assigns to every variable $v \in V$ a value $r \in R$ at certain time point. It is understandable if we associate sensors and actuators with a series of variables such as "battery power", "location" etc. However, simply enumerating the possible combinations of variables does not necessarily constitute meaningful "states", but rather a state that is semantically meaningful is per se a cluster of arbitrarily defined restrictions over variables and relations among variables. While the transition between different states is often associated with a subset of variables $V' \subseteq V$, whose range $R' \subseteq R$. Particularly, the state transition from

$$s_a \xrightarrow{e} s_b$$

where e stands for events that either activate variables outside V' or make values outranging R'. For instances, an integrated sensor transfers from the state "idle" to the state "settingSamplePeriod", which may caused by an input signal trigger the variable "sample period" to be rewritten. Similarly, from the state "idle" to "sleep" was basically due to that the value of variable "battery power" has dropped under the threshold.

State-based interface is said to derive from the theory of Finite State Machine (FSM), which has been widely used to model sequential control logic in digital electronics, where it is defined as a digital device that traverses through a predetermined sequence of states in an orderly fashion, and state is a set of values measured at different parts of the circuit. Most sensor or actuator controlled by digital logic satisfy Finite State Machine (FSM) model [93]. For example, a temperature sensor with its control circuits that measure environment thermal data is an instance of physical FSM. Its states can be graphed as Figure 3.1. The sensor input is analog thermal signal while output is digital temperature data. Its states, e.g. "setting temperature measurement", "setting sampling precision", etc, and

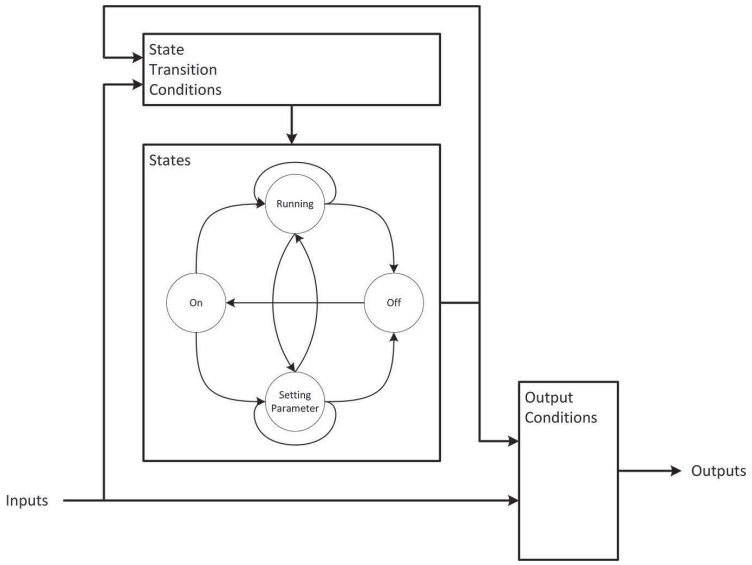


Figure 3.1: A State Machine Model for a Temperature Sensor

state transitions constitude the mapping rules, or say mapping matrix, from input to output. The changes of output reading may result from the changing of input (32℃to 36℃), and/or the changing of states (measurement "Fahrenheit" to "celsius"). The alteration of states is not as intensive as the input/output data, while the frequency of latter may reach as high as hundreds per second. In real practice, it is plausible to cohere low-frequency I/O data with state message together in single XML file, and transmit via standard HTTP operations to obtain maximum interoperability over random heterogeneous systems. However, it entails extra high-speed data channels to transmit realtime, unformatted (usually raw) data stream. In this case, coexisting state messages must convey necessary communication information (e.g. websocket IP address and port number etc.) for establishing transport connections, as well as data schema (e.g. data format, measurement etc.). To assemble proper and meaningful information, state message can be synchronized with I/O data stream, by appointing a flag message contained by I/O data to indicate once the states has been changed.

A more complicated actuator example is provided in Figure 3.2. A smart blind actuator has four states: Full Open, Closed/Half Closed, Pulling Up/Down, and Rotating Slats. Say we want to set the rotation angle of the slats to 5 degrees, first of all we must make sure that the window slats are not totally rolled up like in full open state so that we can adjust the slat angles, as it can be implied from

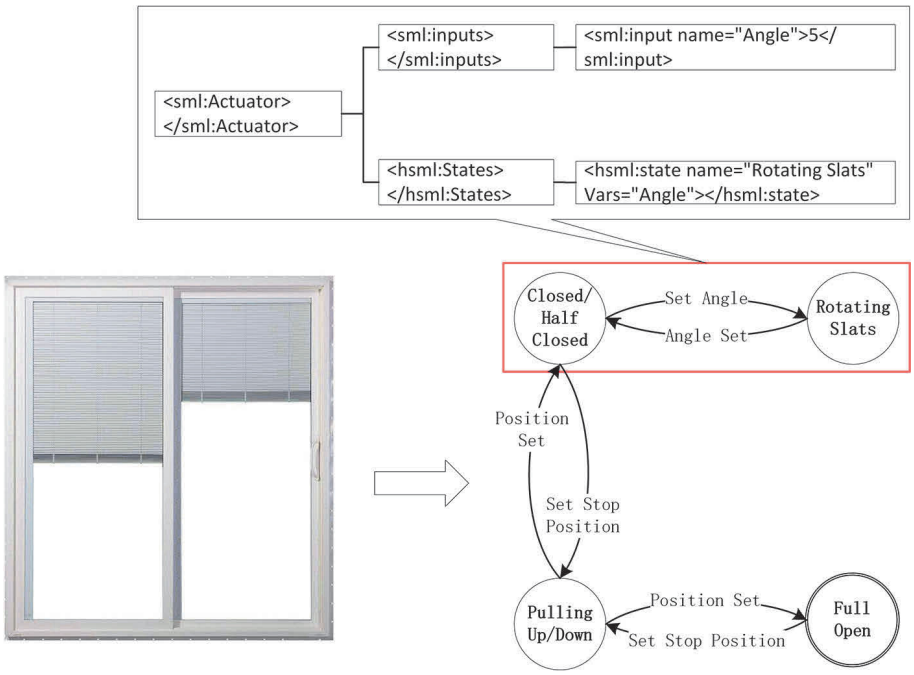


Figure 3.2: A State Machine Model for a smart blind and its State Transfer SensorML Sample

the state machine beside. From "Closed/Half Closed" state to "Rotating Slats" state, we may convey related variables and values in any standard formats like JSON or XML etc., to indicate what final status we want to achieve, and hence avoid directly calling the device-dependent methods to set rotation angles. Here we gave a piece of state message with XML-like encoding (Precisely it should be called "stateML", but we'll leave it to section 3.5), in which the desired rotation angles can be described by the variable "Angle" and the value "5".

3.3 Physical and Virtual State Synchronization

Our motivation to adopt the FSM model is mainly due to the fact that traditional web resource, though maybe is also a projection of some physical object, doesn't need to maintain the same state with its physical source, while the sensor/actuator/IoT resources need to achieve this state synchronization within certain (usually tolerable) error range and delay, because they are supposed to bridge the physical and virtual world. Theoretically, this synchronization can be expressed as a state machine replication process.

Suppose a FSM can be defined as a quintuple $M = (S, \Sigma, \Gamma, s_0, \delta, \omega)$, where S is a finite non-empty set of states, Σ is a finite non-empty set of symbols representing the input alphabet, Γ is a finite non-empty set of symbols representing the output alphabet, $s_0 \in S$ is the initial state, δ is the state transition function $\delta : S \times \Sigma \to S$ in a FSM, and ω is the ouput function $\omega : S \times \Sigma \to \Gamma$.

The corresponding virtual resource can be defined as a copy of the same FSM of the physical resource: $M' = (S, \Sigma', \Gamma', s'_0, \delta', \omega')$, which means for each element of Inputs, States and Outputs in M, there is a corresponding element in M', and given the corresponding Inputs and Sequence of State Transfers, the corresponding Outputs will be obtained. However, the state implementation in M and M' are not necessarily the same. For instance, an "on" state may refer to a logical high level in physical resource, while be implemented as a digital 1 in virtual resource.

Thus, the physical-virtual synchronization can be regarded as an asynchronous state machine replication process. During this process, multiple copies of the identical State Machine begin in the same Start state, perform the same state Transfers in the same order will arrive at the same Target State, and generate the same Outputs from the same Inputs (though may be implemented in different forms). In service architecture, an event (called an "input", "output" or "action") is a standard method to drive a state transfer. In these situations, state machine can be represented by augmenting the state to include the last event. In other words, a transition $s \xrightarrow{\alpha} t$ from state s to state t with event α can be represented as a transition from event-augmented state $< s, \beta >$ to state $< t, \alpha >$, where β is the event that "leads to" s. Based on this definition, we define synchronization between state machine M' and M as follow:

$$s'_0 = s_0$$

For each state transfer $< s, \beta > \to < t, \alpha >$,

$$< s', \beta' > = < s, \beta >$$

$$< t', \alpha' > = < t, \alpha >$$

We argue that it facilitates system composability to automate synchronization between physical and virtual resources. Traditionally, physical-virtual synchronization is handled in 4-step sequential operations, if host service or API that encapsulating sensor functionality is still based on operation: 1) Call (physical) sensor control function. 2) Wait until function return true. 3) Call (virtual) sensor web service. 4) Wait until the service return OK. All the 4 steps must be carried

out in a transaction to certify integrity of synchronization. If any step failed, the whole transaction is incomplete and should be rolled back.

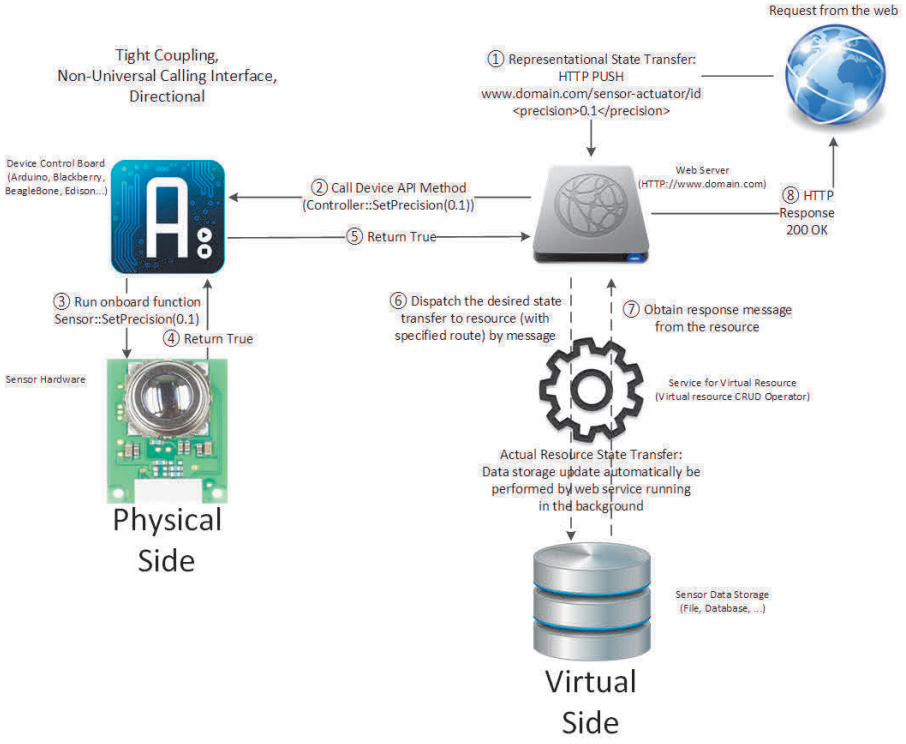


Figure 3.3: Traditional Physical-Virtual Synchronization

While in a complete state transfer oriented mechanism will help to simplify the synchronization process. By delivering state description messages, a bidirectional synchronization service can be achieved to maintain bidirectional messaging coordination automatically and adaptively, as shown in 3.4. In this mechanism, the caller only need to 1) get/set the desired (physical or virtual resource) state transfer other than perform the real synchronization operation, the real synchronization operation will maintained 2) automatically by services running in background. It is comprised of a physical resource service that is used to retrieve data from physical resource and represent data in a state machine based model by wrapping low-level API, a virtual resource service that is used to manage virtual resource (such as database) and respond to web request (better implemented as a state-transfer based service as we will discussed below), and a bidirectional synchronization service that is used to maintain synchronization for both sides

automatically.

Other options also exist, for example: the traditional method that invoking device API directly. However, this kind of methods is tight-coupling, directional solution with non-universal interface, as shown in figure 6, hence does not satisfy our domain requirements.

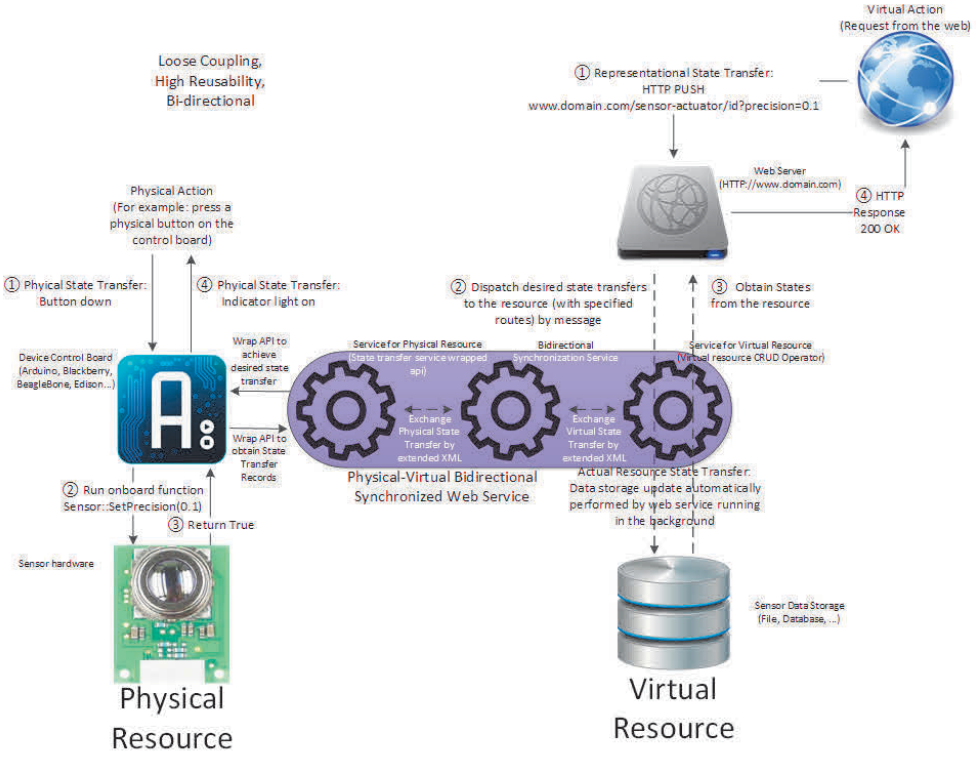


Figure 3.4: State-based Physical-Virtual Synchronization

In addition, multiple physical copies and virtual copies of the state machine can be synchronized to increase fault tolerance. For example: multiple sensors are used to sensing the same physical object, and its value are stored in multiple data storage. In this case, any single point failure will not affect the synchronized resources as a whole if a proper voter mechanism is provided.

Essentially, this sort of structured, interlinked state-based interface reflects the behavior patterns of sensor/actuator resources along with the underlying routes or constraints that one must follow in order to achieve desired states. It provides more sufficient information for resource composition. In addition, it helps to recognize pattern equality or similarity by comparing state machines, thus to

automatically recommend proper service for further processing. For example, a door may be representationally equivalent to a switch button according to their state machines, and thus can be controlled by the same operation logic as shown in Figure 3.5. Last but not least, the state machine model is a natural composable component. Composing or decomposing of resource representation can be easily described as linking or breaking the link of states between state machines.

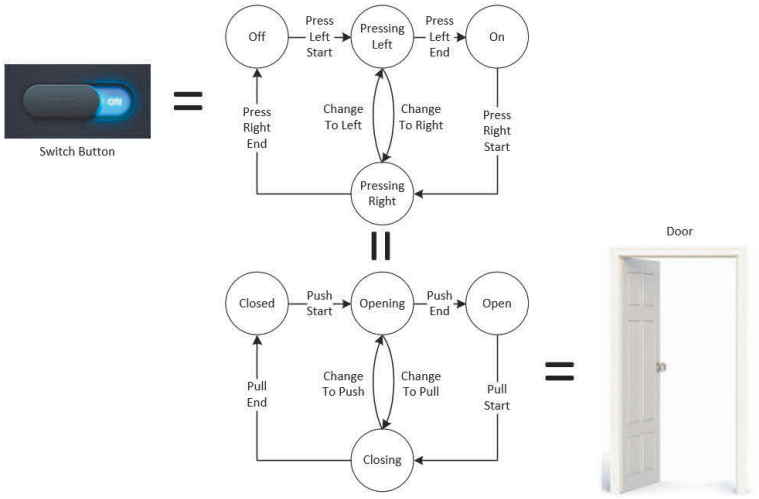


Figure 3.5: Pattern Equivalence between a Switch and a Door

In our composition framework, the minimum requirements for composability is state-based service interface. Optional FSM-based service description file is strongly recommended for extending cross-domain (re-)usage. Notify that the lack of route constraints may entail invalid state transfer attempts. For those IoT host services based on incompatible technology stack, e.g. SOAP, web socket, server-sent event and etc., extensive wrappers are prepared.

3.4 State-Transfer-based IoT Service Composition

Similar as sensor/actuator/IoT resources, researchers have proved that resource based web service can be modeled as FSM as well [94], and simulate physical FSMs like a sensor or an IoT device [95]. Thus, the composition and decomposition among common resources are hence simplified to connect or disconnect FSM-modeled nodes. Specifically, when both web services and IoT resources are equally

exposed as unified state-based services, previous node's state transfer can act as a trigger for the next node's status to change. By establishing this sort of state transfer chain, it becomes feasible to automate the process sequences and complete certain tasks. To establish a state transfer chains under our proposed framework, we are expecting that:

1. A state transfer chain starts from a subset of a composable service state that acts as inputs, e.g. a web camera has a state named "shooting", which may contain a valid variable "focus".

2. The transition of input states will hence trigger the states of the sequential nodes to change according to a set of predefined rules, e.g. a logic gate function or a translation service.

3. A state transfer chain ends with a subset of a composable service state that acts as outputs, e.g. the state "flyingTo" of a irrigation drone contains 2 valid variables, respectively Lat and Lng.

Theoretically, given two composable service node a and b, respectively featured by a set of states S_a and S_b with corresponding collections of variables V_a and V_b. A state $s \in S$ assigns to every variable $v \in V$ a value $s(v)$ in its Range R. Hence we define a link of state transfer e_{ab} between the $v_a \in V_a$ and $v_b \in V_b$ as a relation function of two state transfers:

$$I(e_{ab}) = (s(v_a) \to s'(v_a), s(v_b) \to s'(v_b))$$

If links exist between multiple variable pairs from s to s', then $e_{ab} \in E_{ab}$, where E_{ab} is the set of all links between any two related variables in V_a and V_b respectively.

Given a set of service nodes (at least two), a state transfer chain is defined as a graph of links:

$$G = (T, E, I)$$

Among which, T, E, I represent the set of state transfers from different service nodes, link of state transfers and relation function of the link respectively. I maps each element of E into a $T \times T$ relation space. If $I(e) = (p, q)(e \in E, p, q \in T)$, then we say the state transfer of service node p and q is linked by the relation function $I(e)$, e.g. the state transfer p will trigger the state transfer q according to the relation defined by function $I(e)$.

In designing the IoT service composition mechanism, we have adopted a central orchestration module to dispatch state transfer messages in between physically

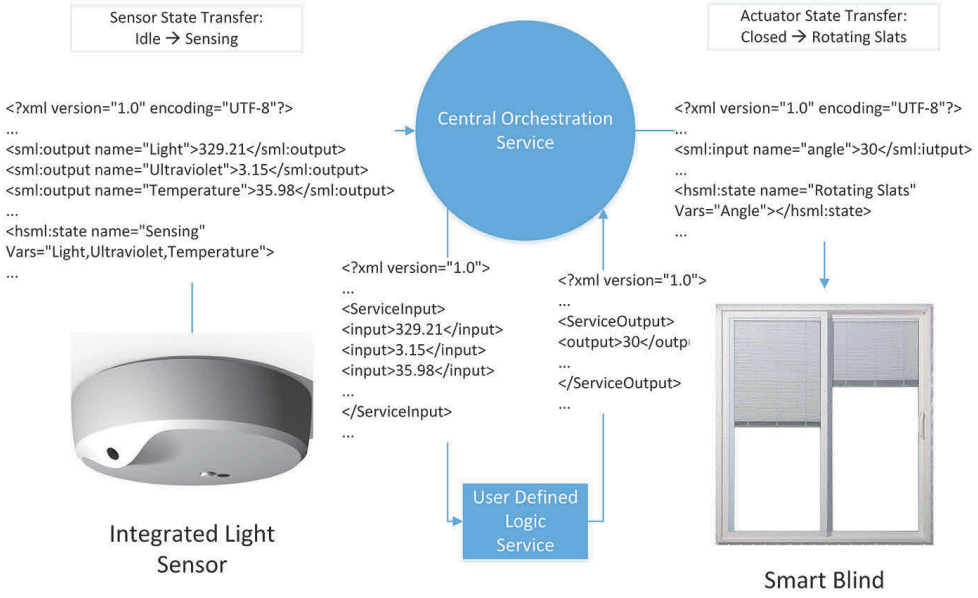


Figure 3.6: A State Transfer Chain Example under Central Orchestration

separated modules as well as arrange executive sequence and logic. As shown in Figure 3.6, a light sensor was used for detecting current illuminance, ultraviolet ray index and indoor air temperature, hence remotely controlling a smart window blind according to user-customized logic, e.g. "roll the blind down when UV index over 6 OR indoor temperature higher than 30 degree Celsius". This seemingly simple composition involves four main modules: (1) a light sensor, whose state transferred from "idle" to "Sensing" is translated to corresponding light data by (2) A user-defined logic service, further triggered the state transfer of (3) window blind from "Open" to "Rotating Slats", and (4) the central orchestration service is in charge of all the messaging and control flow.

Specifically, in our proposed solution, state transfers are expressed in a set of variables that stored and transmitted in standard-formatted files, which are supposed to contain full information that a single request needs to be completed, so that each connection initiated between any two nodes within a state transfer chain stays independently from the previous and latter connections. We consider the statelessness as an indispensable property in order to maintain the stability and scalability of our proposed system, particularly in a distributed computing environment.

3.5 StateML: A Unified Resource Representation

3.5.1 Hybrid State-based Service Interface Description

In this section, we propose StateML for expressing the aforementioned FSM-modelled service interface in a machine-readable format. It is a XML specification combining both SensorML [96] and SCXML [97]. In Figure.3.7, we presented the beginning part of a sample sensor description file, which contains a united name space of SensorML and StateML. It should be noted that, $xmlns : xlink$ allows the usage of XInclude [98] for merging multiple XML files. This is specifically useful when multiple resources share similar metadata and finite state machine models. The common part can be referred as a external html link, e.g. $< sml : outputs\, xlink : href =$ "$http : //resource.example.com/sensor/commonDescription$" $>$ and then merged into a complete XML file.

```
<?xml version="1.0" encoding="UTF-8"?>

<stateml:StateMachine>
  xmlns:stateml="http://www2.kmd.keio.ac.jp/~ruowei.xiao/stateml" <!-- StateML name space -->
  xmlns:scxml="http://www.w3.org/2005/07/scxml"
  xmlns:sml="http://www.opengis.net/sensorml/2.0"
  xmlns:swe="http://www.opengis.net/swe/2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xsi:schemaLocation="http://www.opengis.net/sensorml/2.0 http://schemas.opengis.net/sensorml/2.0/sensorML.xsd"
```

Figure 3.7: Namespace in SensorSample

In stateML, we adopt SensorML for specifying basic data schema with a full set of variables and their domains. SensorML is the standard initiated by Open Geospatial Consortium (OGC) with the aim to provide standard models and an XML encoding for describing sensors and measurement processes. The most updated version now is SensorML 2.0, which came out in 2012. According to SensorML 2.0 specification, a typical physical component description may concern general system description (device identifier, keyword list), identifiers (manufacturer, model information), classifiers (sensor type and intended application), characteristics (device physical properties like weight, length and electrical requirement), capabilities (sample period, output interval etc), input lists (expected inputs), output lists (expected outputs, measurement) and parameter lists (response parameters like relative response curve).

An integrated environmental sensor can be defined as a physical component

```
<!-- Inputs = Observed Properties -->
  <sml:inputs>
      <sml:InputList>
        <sml:input name="temperature">
            <sml:ObservableProperty definition="http://sweet.jpl.nasa.gov/2.3/propTemperature.owl#Temperature"/>
            </sml:input>
            <sml:input name="Air"> ··· </sml:input>
            <sml:input name="Sound"> ··· </sml:input>
      </sml:InputList>
  </sml:inputs>
```

Figure 3.8: Input List in SensorSample

and its input/output variables are respectively described in Figure 3.8 and 3.9. A full description is supposed to also include information like device, id, location information and etc., but the sample files provided in this dissertation have omitted some of the sections for brevity. A more complete version can be found in Appendix A.

```
<!-- Observed Property = Output -->
  <sml:outputs>
      <sml:OutputList>
        <sml:output name="IntegratedSensorStream">
            <sml:DataInterface>
                <sml:data>
                    <swe:DataStream>
                        <swe:elementType name="environmental_data">
                            <swe:DataRecord>
                                <swe:field name="Relative Humidity">
                                    <swe:Quantity
                                    definition="http://mmisw.org/ont/CUAHSI/AtmosphericHydrologicCore/relativeHumidity">
                                        <swe:uom code="%RH" />
                                    </swe:Quantity>
                                </swe:field>
                                <swe:field name="Temperature"> ··· </swe:field>
                                <swe:field name="Pressure"> ··· </swe:field>
                                <swe:field name="Sound Intensity"> ··· </swe:field>
                                <swe:field name="Battery voltage"> ··· </swe:field>
                            </swe:DataRecord>
                        </swe:elementType>

                        <swe:encoding>
                            <swe: TextEncoding tokenSeperator="," blockSeparator=" "/>
                        </swe:encoding>

    <!-- a Real-Time-Protocol (RTP) server that continues to stream real time measurements -->
                        <swe:values xlink:href="rtp://myServer.com:4563/sensor/02080"/>
                    </swe:DataStream>
                </sml:data>
            </sml:DataInterface>
        </sml:output>
    </sml:OutputList>
  </sml:outputs>
</sml:PhysicalComponent>
```

Figure 3.9: Output List in SensorSample

The input/output datafields of a physical component are generally equivalent to the entry/exit of state machine of the same device. The input list specifies observable properties or phenomena, while the output list mainly describes data that the device observed, its measurement and values. In the example,

output lists included relative humidity, temperature, pressure, acoustic intensity etc. Usually, last measurement values are expressed in the $xlink : href$ attribute of $swe : value$ by an html link, e.g. $< swe : values\ xlink : href = \text{“}http : //myServer.com/sensor/node1\text{”} >$.

But it is also possible to specify realtime streaming protocols, like Real-Time Protocol (RTP) in the example. SensorML 2.0 claims that there are various other protocols that could be supported by a SensorML description, and supposed to be described in the $sml : interfaceProperties$ element of the $sml : DataInterface$ object.

While the SCXML part of descriptions, on the other hand, describes the corresponding state chart that embodies the cluster of arbitrarily defined restrictions among variables. State Chart XML (SCXML) is a general-purpose, event-based state machine language standard proposed by World Wide Web Consortium (W3C) in 2015. In SCXML, a state machine is commonly decided by a triad of “state”, “transition” and “event”. Each state contains a set of transitions that define how it reacts to events, which can be generated by the state machine itself or by external entities.

Particularly, a state may contain multiple nested children states, as known as either “compound” state or “parallel” state. The former refers to the kind of state that when it’s active, exactly one of its children states is active. While the latter refers to that when the parent state is active, all of its children are active. Thus, by adopting this nested structure, it is able to depict complex control logic of especially multi-components IoT items.

In Figure 3.10, we provided a FSM model of the sensor sample. For brevity, we listed only the SCXML part of the corresponding stateML description in 3.11. It starts with an initial state “idle”. To specifically point out that, internal events that trigger states to naturally transfer are not explicitly specified, e.g. when state “SettingSampleInterval” is over, it will continue skip to state “idle” without external intervention. And events that trigger state transition are always bound to the changes of significant variables and their value, which are supposed to be declared in the previous SensorML part in order to state their value range.

There are two different types of events: The “Internal” events that are only observable, e.g. the transition from “Idle” to “Sleep” caused by natural power attenuation within the system; And “External” events that triggered by external operations, e.g. the transition from “idle” to “settingOutputInterval” by the operation of changing the variable “OutputInterval”. Input and Output are two

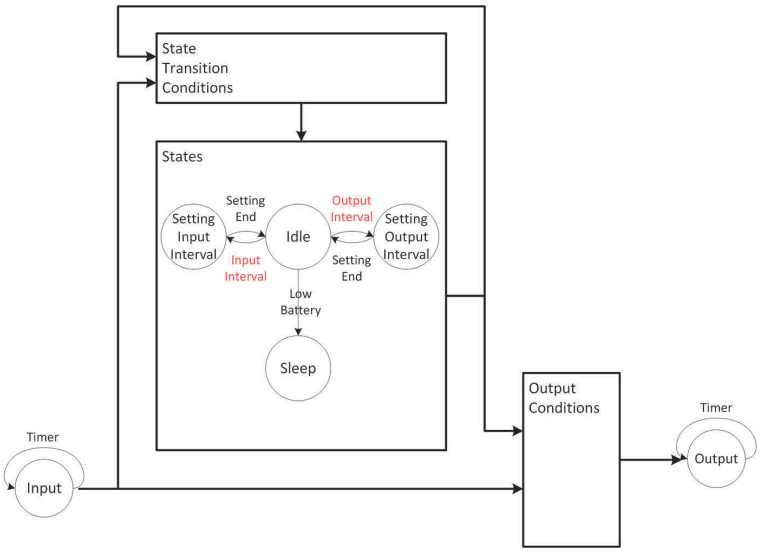


Figure 3.10: Sensor Finite State Machine

special standalone states, which means they cannot transit to other states but themselves. Compound state and parallel state can be used to decribe more complicated behavior patterns of IoT services.

```
<!-- Sensor as a State Machine Description -->
<!-- We don't specify internal events that trigger state to naturally transfer, e,g, when sensing state is over,
     it will continue to outputting state without user intervention. -->
<!-- Events that trigger state transition are always binded to significant variables and their value changes.
     And the variables must be described in the SensorML in order to state their range and I/O type. -->
  <stateml:states>
    <scxml:state id="SensorSample" initial="idle">
        <scxml:state id="idle">
            <scxml:transition event="Battery voltage: U_INT_8" type="internal" target="sleep"></scxml:transition>
            <scxml:transition event="outputInterval: U_INT_8" type="external" target="settingOutputInterval"></scxml:transition>
            <scxml:transition event="sampleInterval: U_INT_8" type="external" target="settngSampleInterval"></scxml:transition>
        </scxml:state>
        <scxml:state id="settingSampleInterval">
            <scxml:transition target="idle"></scxml:transition>
        </scxml:state>
        <scxml:state id="settingOutputInterval">
            <scxml:transition target="idle"></scxml:transition>
        </scxml:state>
        <scxml:state id="sleep"></scxml:state>

        <scxml:state id="output">
            <scxml:transition event="Temperature: S_INT_16; Relative Humidity: S_INT_16; Pressure: S_INT_16; Sound: S_INT_16"
                              type="internal" target="output"></scxml:transition>
        </scxml:state>
    </scxml:state>
  </stateml:states>
</stateml:StateMachine>
```

Figure 3.11: State Chart in SensorSample

There are a few unique design principles in our practise regarding the usauge of SCXML, which differs from the original standard and are worthy of attentions.

Firstly, the SCXML is for service interface descriptions that greatly depend on what functionality and in what granularity the service developers want to expose, and its style even for the same device sometimes will vary from case to case, rather than strict device's internal mechanism. State and its transitions triggered by internal events, which is neither "operable" nor "observable" to external entities, are supposed to stay transparent.

Secondly, it is recommended that to specify "events" by the variables predefined by SensorML in a key-value pair manner. In most occasions, this method conveys imperative messages by designating key variables related to the desired target state. But if complicated computing, e.g. conditional judgment, timer invocation etc., is inevitable, it should follow specific instruction provided by the service developer.

Last but no least, all SCXML statements within our framework will be exempted from being parsed as executable code line by line as suggested by its original proposal. Instead, it is supposed to stay independently from the actual execution. Synergizing with SensorML, the overall service description file will be uploaded to and stored in some resource management server when a IoT host service is registered. And once the service is requested, related description file will hence be obtained and parsed so that the request initiator, who may be a human user or a service agent, will be informed about 1) what state of the resource can be obtained and 2) what information is required and how to actively switch current state to another.

As concluded in Chapter 2, IoT services obeying these design principles are in accordance with REST paradigm and exhibit very strong similarity with traditional web services, which greatly facilitates the interoperability and compatibility of web components as well as resuability of existing web services. Moreover, customizable host service implies that service developers have their own control over concealing or exposing internal mechanism and functionality of IoT resources at their desired granularity. This kind of unified resource representation conveys adequate information by for both human and machine users to remotely operate and interact with IoT services, but also potentially lays the foundation for future process automation.

Sequential tasks involving multiple state transfers, establishment of complex state transfer chain and some other issues will be discussed in details in the following section.

3.5.2 State-Transfer-based Messaging

In section 3.5.1, we have recommended service developers to adopt stateXML for uniform service interface description. To simplify the management of state information, it is also recommended to transmit event message between nodes using the same format. One of the major tasks of service composition is to fetch and coordinate designated Web and IoT services, and assemble them into value-added composite services according to predefined linking rules. Therefore, we adopted a central service orchestration to analyze user-defined task logic and manage all the state transfer messages between different IoT service node. Detailed implementation of central service orchestration will be discussed in next chapter.

To better illustrate state transfer based service orchestration, we will still use the same sensor example and start from the simplest state transitions within same node, say, to adjust the outputting interval from 2.58 to 3 seconds. For brevity, the namespace prefixes are omitted:

```
<state id="idle">
    <transition event="outputInterval:3" type="external" target="settingOutputInterval">
    </transition>
</state>
```

This piece of stateML message describes the external event that triggers the state transition from *idle* to *settingOutputInterval* with a parameter named *outputInterval* : 3. If using typical RESTful interface, the parameter can be passed to resource server by using standard HTTP GET method, e.g. `http://www.resource.com/sensor/node1?outputInterval=3`. Event messages sent by central orchestration service will be parsed by host service and trigger corresponding API/drivers function that set data output interval to be executed.

Tasks that consist of sequential control commands require more than single hop state transitions, e.g. to make a drone to take off until it reaches 2 meters above ground, then fly forward at the speed of 1 meter per second as well as down at 0.2 meters per second. One critical issue here, is to maintain the stateless status of the server side, i.e. the drone in this case. Therefore, the client, i.e. the service orchestration service, instead will establish a state transfer pipe to guarantee sequential execution, and provide all necessary data to complete every step of state transition in single request. Accordingly, the IoT resource should ignore the requests initiated by other clients until current task is completed, and keep its state visible to client during the whole procedure so that central service

orchestration is able to be notified if each step of the state transition is successfully completed.

```
<state id="takingOff">
    <transition event="height:2" type="external" target="hovering"></transition>
</state>
<state id="hovering">
    <transition event="forwardSpeed: 1; downSpeed: 0.2" type="external" target="flying"></transition>
</state>
......
```

Now that we have the two previous examples as our basis, finally the state transfers from different resources will be assembled to form a complete state transfer chain. Suppose a common smart agriculture scenario, sensors are deployed to detect the soil humidity and once the relative humidity is lower than 20 percents, an irrigation drone will be notified and sent to the spot. The purpose of resource composition here, is to use the sensor's humidity data and geographic location to trigger the drone's actions by establishing a control logic based on specified state transfer chain.

```
<state id="sensing">
    <transition event="humidity: 17.6; coordinates: {35.554498, 139.6485728};"
                type="internal" target="sensing"></transition>
</state>

<state id="hovering">
    <transition event="lat: 35.554498; lng: 139.6485728" type="external"
                target="flying"></transition>
</state>
```

When central service orchestration received the state update notification from source node each time, it will call and run through threshold checking component, may it be an internal function or external web service. If the condition is true, sensor's geographical information will be delivered to the irrigation drone and set the drone's status to complete the following transition from "taking off" to "flying" to the predefined destination.

The composition output is supposed to be an value-added service comprises several existing resources, for example: a temperature alarm service built upon a thermal sensor, a threshold checking service and an actuator host service. This kind of composition result is supposed to be composable, and can be further integrated into more complicated services, which can be proved by related researches that the cascade composition of two finite state machines is still a finite state machine [99, 100].

3.6 Summary

In this chapter, we discussed the general approach for establishing an IoT service composition framework based on the concepts of "state" and "state transfer", which were able to improve service composability compared with its operation-based counterparts.

Specifically, a state of a system was its condition at a particular point in time that could be defined by a set of variables and specic values. It was understandable if we associated sensors and actuators with variables such as "battery power", "location" etc. However, simply enumerating the possible combinations of variables did not necessarily constitute meaningful "states", but rather a state that was semantically meaningful was per se a cluster of arbitrarily defined restrictions over variables and relations among variables. While the transition from one state to another could be defined by a triad of initial state, target state, and triggering event that could be expressed by the variance of significant variables and/or values.

In proposed composition framework, we stipulated that a composable IoT resource must be addressable and accessible via a unique resource identifier (URI), i.e. URI-dereferencable, and the service interface it exposed must be based on states. This kind of state-based interface could be expressed by a Finite-State-Machine model. Then a composition task might consist of one or more state transfer chains that started and ended with a subset of a composable service states, while the state transition of previous node triggered its subsequent node's state to transfer successively according to predefined linking rules. Accordingly, we recommended service developers to adopt StateML, a unified resource representation in machine-readable format, specifically for state-based service description and event messaging.

By adopting proposed approach, it facilitated system composability and helped maintain black-box models and loose component coupling, as well as enabled automatic control logic among components.

Chapter 4
IoT Service Composition Framework: HSML

4.1 Servitization

In IoT domain, IoT host services have been proposed with the aim at enabling seamless interoperability and open accessibility of sensor/actuator/IoT nodes from different vendors via uniform management, with the interposition of an abstraction layer between the application logic and device drivers/APIs [9, 101, 102]. It intends to virtualize physical devices and their functionality into URI dereferencable services with general interfaces, which are also supposed to be compatible with existing Web services. The proposed framework in this research has premises about the particular way how servitized IoT nodes are introduced into proposed framework, however, it does not have any concrete specification on either the servitization technology or methods, as the procedure is usually highly device dependent and vendor specific.

Nowadays, we have witnessed many business IoT solution providers including Amazon (Device Shadow), IBM (Bluemix), Google Cloud IoT etc., competitively offer from-device-to-service solutions that equipped with REpresentational State Transfer(REST) style service interfaces. And though not compulsory, currently most COAP-based IoT services also appear with state-based interfaces. Still, we will discuss three mainstream servitization approaches, and implement several host services as examples. But please note that, it should not be limited to the discussed methods in real practise, as long as the service interface fulfills the aforementioned requirements.

According to different locations that IoT host services are launched, mainstream servitization can be divided into three categories: **local servitization**, **edge-based servitization** and **cloud-based servitization**, as shown in in Figure 4.1. Local servitization allows IoT resources with build-in host service to

directly connect to the Web. In this case, upper-layer applications request services by accessing the device directly. It usually requires the device manufacturer to provide a full-tier solution to traverse TCP/IP stack, or its resource-constraint counterpart, e.g. 6LoWPAN, COAP etc, which also suggests the device node must possess certain amount of computing resources in order to accomplish this kind of local servitization. On the other hand, edge-based servitization and cloud-based

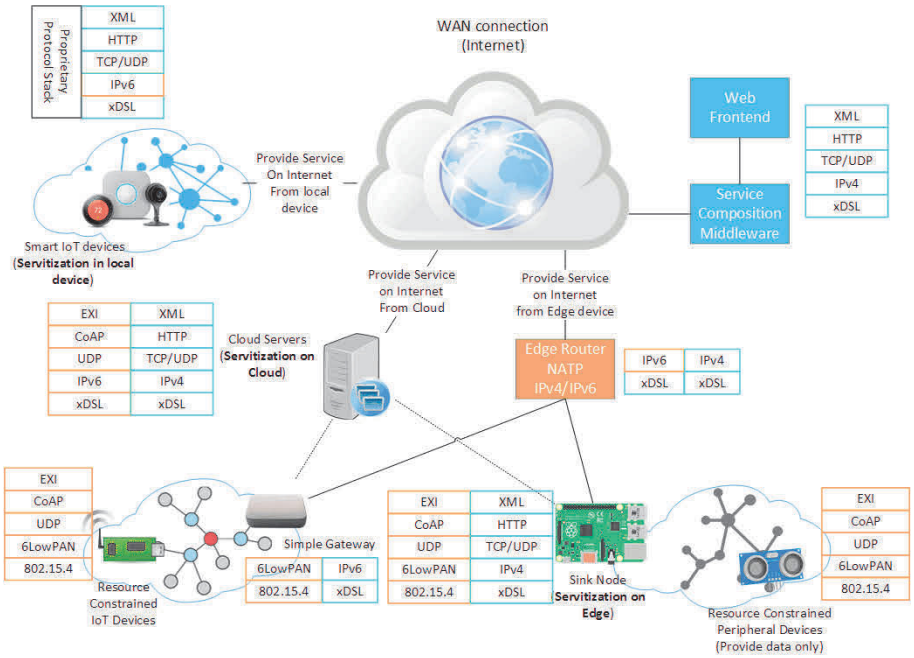


Figure 4.1: Three Different Types of Mainstream Servitization

servitization are considered propitious to large-scale, low-cost IoT nodes with less computing capability. In the former case, host services are located in some sink nodes which collect data from peripheral devices and perform necessary computing operations. Thus, service interfaces are separated from actual resources and usually rely on edge devices, such as a Raspberry Pi, to provide indirect accessibility of IoT resource nodes.

Similarly, the sink node can either be replaced by, or further export data to a central cloud platform, such as the business IoT cloud solutions we mentioned. Cloud-based servitization provides one or more types of service interfaces and pay-as-you-go computing power, which makes it a better solution specifically for cross-organizational, elastic application scenarios.

Though out of our research scope, we would like to give a concrete example of what a typical IoT host service may look like. The following is a fragment of NodeJS code that runs on an actual server to host a drone. Full script is available at Appendix C. Suppose the drone has a URL: `http://resource.example.com/drone`, all of its functions is accessible via corresponding path names, e.g. `http://resource.example.com/drone/land`, and can be requested by using sheer standard HTTP operations: GET, POST, UPDATE and DELETE. Once server-side application detects a POST request with path name "land", it will then call the device API functions stop() and land() to accomplish desired operation. If drone's successfully landed, server will response with a status code "200", which means "OK", as well as a JSON object "currentState" to inform the client of drone's current state.

```
app.post('/land', function(request, response){
    client.stop();
    client.land();
    currentState = 'land';
    response.status(200).json({state:currentState});
});
```

It can be inferred from this example that a host service works as an abstract layer that conceal native device functions, and map them into the state-based interface. And by separating the underlying mechanisms from the task logic and application atop, even if hardware got replaced by another vendors device, or the version of device APIs updated, as long as the host service interface stay the same, the upper-layer application logic can work as usual without readjustment. Another advantage lies in the re-usage of existing services and legacy functionality. Even if the existing service does not have state-based service interface, we are able to effectively control the customization cost by adding just one more layer of host service as an adapter, while maintain existing structures.

Though some device manufacturers already provide their products with RESTful interfaces, in that case their device APIs are regarded equal to a host service. But most of the time, you may need a host service that to translate vendor-specific API functions to language-independent, state-based web service interface. We actually expect that the servitization of IoT resources is beforehand taken over by service developers, research communities, device owners, or even some hardware manufacturers. And before resources enter our proposed framework, state-based interfaces as shown above must be prepared to allow further composability. Beyond, it is strongly recommended service developers to provide a standard service

description along with the host service using, e.g. StateML discussed in last chapter, for the purpose of host service registry, query and acquisition that better support the sharing of resource information among different stakeholders.

Sometimes, service developers may want to provide multi-prong service accessibility in response to diverse actual demands. For example, IoT developers can choose to request realtime, high-precision data by directly accessing a physical sensor via LTE or 4G network, which may hence generate relatively expensive charge. Or he/she can instead request stored data from the corresponding virtualized sensor at a lower service cost. Thus, a well-servitizated IoT resource allows service providers to flexibility adjust its service quality according to different stakeholder interests, cost and authorities.

No matter whichever approach is selected, IoT resources are supposed to be equally viewed as homogenized services from external perspective after servitization, and can be uniquely addressed and accessed through standard Web messaging. Networked sensor systems, e.g. wireless sensor network, can share single host service at the sink node, and data from each sensor node can be streamed via URI path. For IoT resources that need to have dedicated address, one of the possible solutions is offered by IPv6 standard, which provides a 128-bit address field, thus making it capable to assign a unique IPv6 address to any possible node in the IoT network. The aforementioned standard 6LoWPAN, which is an established compression format for IPv6 and UDP headers, can be transparently translated from/to IPv6 by deploying a board router at the edge between IoT network and IPv6 network. While the conflict may exist between IoT nodes and IPv4-only hosts, it can be addressed by a few proposed methods including v4/v6 Port Address Translation (v4/v6 PAT), v4/v6 Domain Name Conversion, and URI Mapping etc [44].

Furthermore, service monitoring and tracking facility may need to be presented after servitization in order to deal with the inherently unreliable nature of IoT services, that cannot be assumed “always on”, as mobile-powered ones may go offline in one location and turn up again somewhere else, and the availability of some services may swing steadily in an unpredictable way [103].

4.2 General System Architecture

To provide a general system overview, proposed IoT service composition framework comprises a few core modules, as shown in Figure 4.2. Firstly, A **web**

application development toolkit, which can be either based on sheer graphic elements (GE) or domain specific languages (DSL), to acquire customized requirements and return the composition output. Here we provides a domain specific language named Hyper Sensor Markup language (HSML) to enable IoT developers specifically describe single IoT node's access, process logic all the way to visualization effects, as well as linking rules among multiple service nodes. The usage of HSML will be introduced in details in coming sections. The toolkit hence consists of three modules: 1) A file uploader, for IoT developers to upload images and local data file like CSV and XML file; 2) An HSML text editor for IoT developers to input and edit HSML texts and 3) A geo-visualizer to help IoT developers debug and tweak data visualization effects. This component is mainly developed by JavaScript and run on web browsers of Client side (support Chrome, FireFox and Safari. Incompatible with IE).

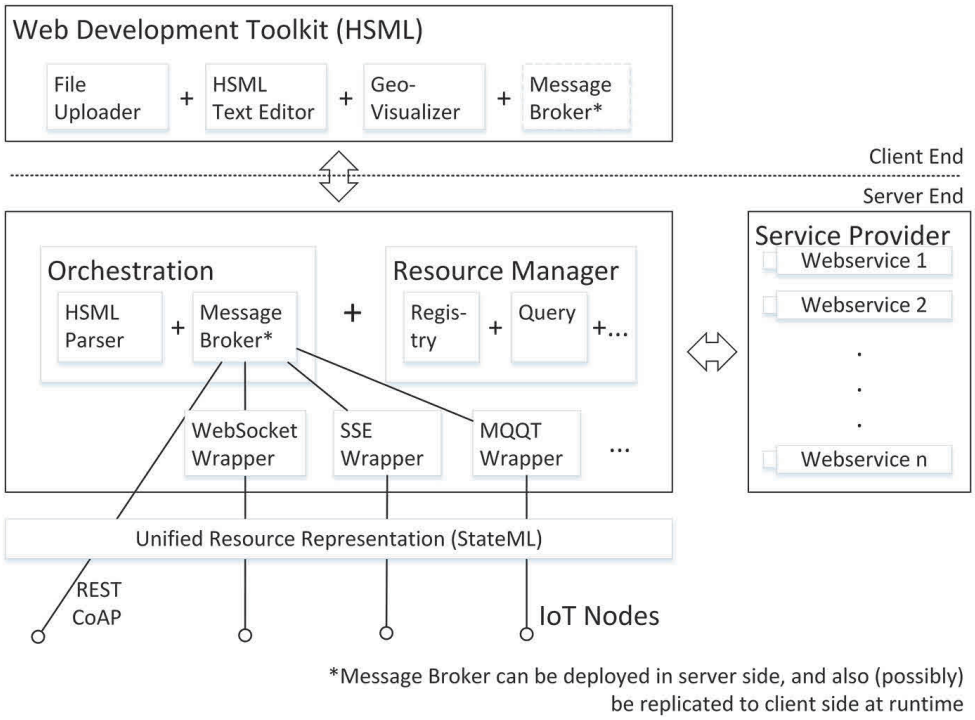


Figure 4.2: System Components

Secondly, a **central service orchestration** is in charge of analyzing received user-defined task logics and coordinating service nodes. It is comprised of 1) a HSML parser along with 2) one or multiple message brokers, while the former

parsed submitted HSML texts into corresponding attributes and values, according to which the later will be constructed and configured. The central orchestration service, which usually locates at composition server clusters, receives and analyzed composition requests initiated by one or multiple clients. It will further establish service and linking route tables, configure message brokers and pass necessary information that needed to accomplish a specific composition task.

Message brokers will then take over delivering and receiving state messages between different nodes, which can be duplicated and deployed on distributed environments according to different application scenarios. It is the actual agent to handle the request/response between service nodes and transmit state transfer messages from previous node to the next according to predefined linking routes and rules. Message broker helps maintain the communication in between sequential nodes based on the transitions of state. The overall central service orchestration module is developed by NodeJS and can be run on most mainstream server systems. While in some deployment case that message broker is allowed to be replicated and run on client ends, NodeJS can actually be replaced by JavaScript or Python etc. The reason why we chose NodeJS was because it shared a similar syntax and grammar with our front-end language: JavaScript. Also, it ensures compatibility with other mainstream IoT middleware including IBM's Node-Red.

The **resource management module** contains functional components such as service query and register etc, which developed by a combination of NodeJS, JavaScript and MongoDB in our actual implementation. As already mentioned in Section 3.5, we strongly recommend service developers who wish to open their IoT service access to other IoT developers, to provide uniform resource descriptions to for resource registry and query so that it can better support the sharing of resource information among different stakeholders. The resource management server can be deployed either together with orchestration service, or independently on specific resource servers to enable flexible resource import from external organizations and institutions.

Finally, for IoT services whose programming interface and/or driver does not in compliance with the requests of FSM model, as we stated in previous sections, a couple of **wrappers** are ready for encapsulation and interface translation. At current stage, besides REST (CKAN/ DKAN), COAP (Eclipse Californium), we have also prepared compatible wrappers for Web Socket (Socrata), AJAX (OpenSensor.io), server-sent event (OGC), MQTT, as well as data formats like XML, JSON and CSV. Service developers can also develop their own wrappers.

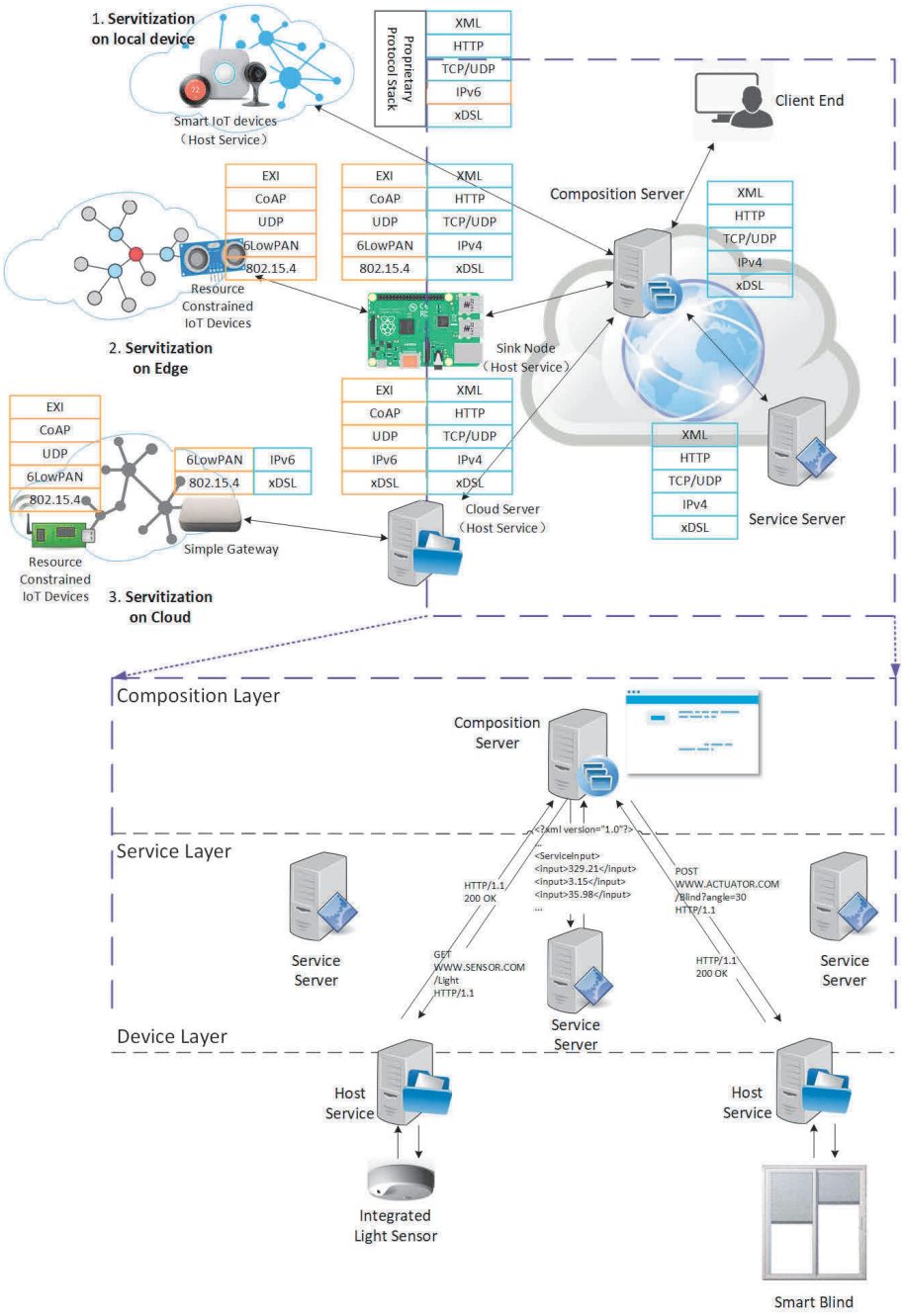


Figure 4.3: A Three-Layer System Architecture Layout

Generally, the typical deployment of our proposed system within a distributed computing context is a three-layer system architecture. The top-most level is

the **Composition Layer**. It fetches and coordinates designated services and resources, and assembles them into new value-added composite services according to user-customized rules and logic control. Therefore, a concise user interface must be provided in this layer.

Service Layer manages a pool of atomic building blocks that spread through cross-organization server clusters, to further process data from lower layer, carries out calculation and provides outcome to upper layer. Each service is supposed to maintain its self-description files and maintain available state transfers information.

Device Layer is supposed to collect data from edge devices or gateways, aggregate and wrap them up into host services with uniform interfaces, which retrieve data from local devices (local servitization), edge sink nodes (edge servitization, usually instant data) or cloud servers (cloud servitization, usually historical data). Developers who provide host services are also supposed to offer necessary information for other IoT developers to interact with the service node, either by organizing and exposing related functionality in the way of hypermedia, also known as HATEOAS (Hypermedia as the engine), or by registering certain interface description files for external services to query over and access.

Multiple device networks which may be set up and managed by different organizations and institutions can be equally introduced into the same deployment. An overall system layout inside distributed computing environment is presented in Figure 4.3.

4.3 Web Development Toolkit

4.3.1 HSML Syntax Paradigm

To evaluate the feasibility, usability and other desired properties and provide a typical implementation of our proposed framework, a web development toolkit for IoT web application: Hyper Sensor Markup Language (HSML) was hence invented. To pay specific attentions, we intentionally to use the term “HSML” to indicate **both** the underlying IoT service composition architecture, as well as the domain specific language that used for constructing composition task logic.

Three possible implementation approaches had been discussed: 1) Graphic Elements (GE) based, 2) Domain Specific Languages (DSL) based, or 3) a combination of both. The former utilizes graphical icons to represent application domain

functions and information specific functions, and give a visual face to underlying web resources. It allows novice developers without adequate programming skills to create customized resource compositions via simple and intuitive user interactions. On the other hand, DSL based approach provides IoT developers with a terse programming language with common syntax and semantics regarding a particular domain [104]. The trade off between kick-start barriers and expressiveness has hence facilitated the hybrid paradigm of GE and DSL.

In our implementation, we have adopted the DSL approach with a HTML-like syntax due to: 1) HTML is widely accepted not only by technical community, but also within design and business domains, which results in relatively lower learning barriers and a potentially larger user group. 2) HTML, as a fundamental component of current web technology stack, provides higher compatibility when integrated with most state-of-art web technologies. 3) HTML shares a consistent syntax with XML, while the later one is usually used as the description language of many current semantic sensor web standards, including SensorML, SSN and etc.

More precisely, we have designed three kinds of descriptors to convey main subjects involved in resource composition, respectively:

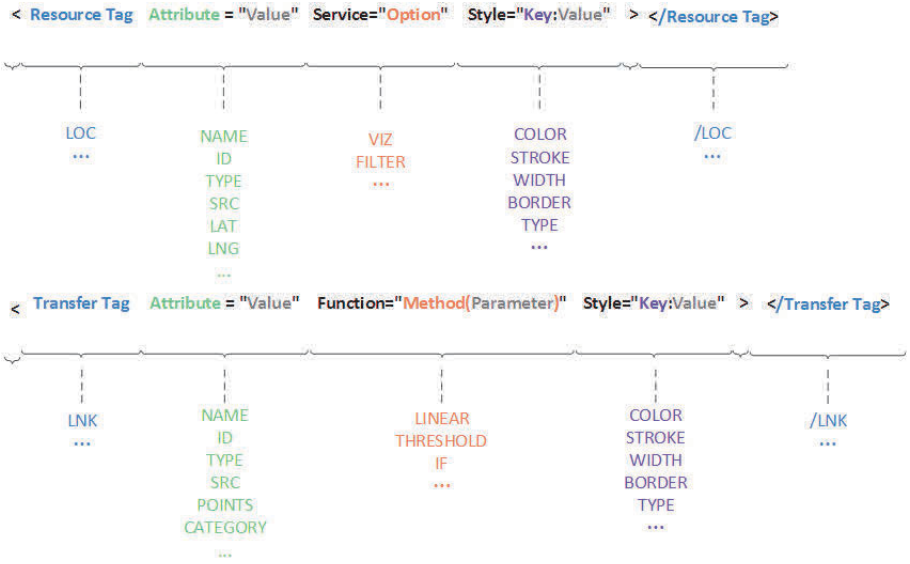


Figure 4.4: HSML Syntax Paradigm

1. **Resource Descriptor**, for loading and importing composable sensor/ actuator resources. Typical usage like: $< loc\ src = \text{“}www.example.com/sensor$

" $><$ /$loc>$, among which compulsory attribute src is required to specify the URL of sensor/actuator resources. Other optional attribute like *type*, *name* etc can be used to described extensive information about a resource.

2. **Service Descriptor**, for associating specific service components with imported resources. Fox example, to create a bar chart from sensor data, we use: $<$ *loc src* = "*www.example.com/sensor*" *viz* = "*bar*" $><$ /$loc>$, among which an external data visualization service component is inputted and mapped to an attribute called *viz* further associated to the sensor resources. In consideration of system security and other related issues, we dont provide IoT developers with open interface for importing external services for now. Service components can be introduced as library or inlined directives by administrator who deploy and manage the overall HSML system only.

3. **State-Transfer Descriptor**, for linking state transfers of two or more resources together according to designated rules. A typical usage like using a temperature sensor to control an alarm. When temperature crosses the predefined threshold, for instance 100 centigrade, the alarm will be turned on. Corresponding HSML expression will be like $<$ *lnk function* = "$THRESHOLD(alarm.on,\ sensor.temp, 100)$"; $><$ *lnk* $>$, among which *alarm* and *sensor* are declared web resources, and *threshold* will be a boolean function stipulating relationships between the state transfers of two resources.

Two different composition types can actually be implied from above: **implicit composition** and **explicit composition**. Generally, in the situation of implicit composition, we compose a resource with one or multiple services without explicitly appointing a state-transfer descriptor. The service nodes are syntactically "attached to" a resource descriptor as attributes, the executive sequence of which are either designated by the default composition logic or priority levels of each service components. On the contrary, in explicit composition we use the state-transfer descriptor like $<$ *lnk* $>$, to explicitly specify one or multiple state transfer chains that consisted of at least two resource states and related linking rules.

Figure 4.5 shown the file uploader of HSML web API, where IoT developers can upload files, e.g. csv files, xml files, images etc, to the composition server. Static data repositories, batch processing script or advanced programming code can be similarly inserted via URL into HSML. Next interface is web based HSML editor

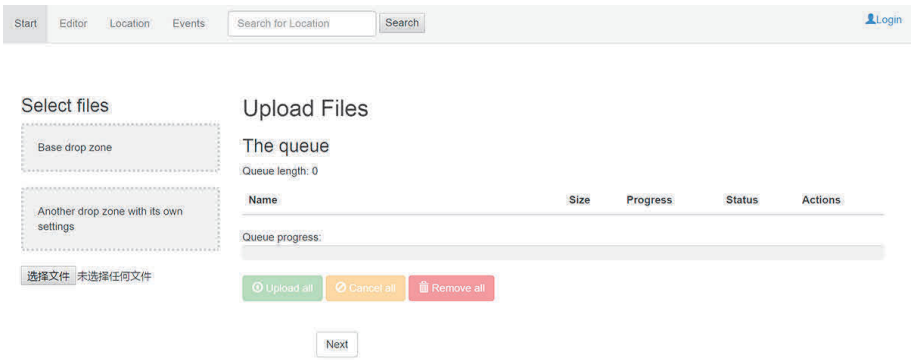


Figure 4.5: File Uploader of HSML web API

as shown in Figure 4.6. Developers can edit, execute and view the composition result as well as debug their HSML in an on-the-fly manner.

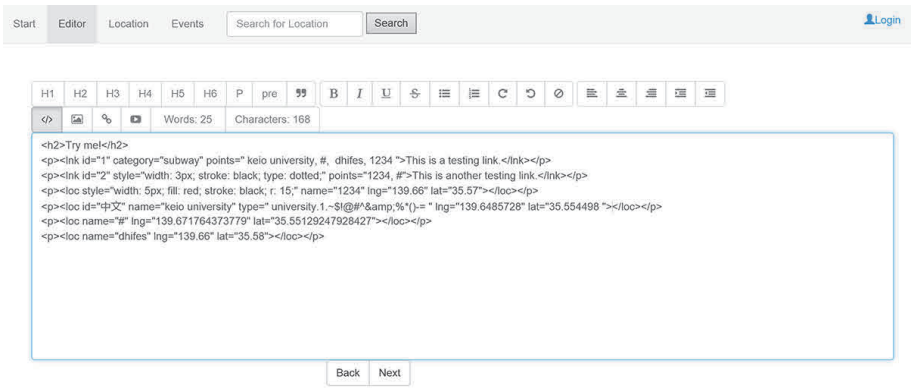


Figure 4.6: Web HSML Editor

In addition, a geo-visualization interface based on web map (Google map in current stage) is also part of the API. Developer can type location keywords to query and insert specific locations into their HSML texts. Compiled state transfer results and visualized data will be mashed up and integrated into the map interface. Each resource descriptor $< loc >$ will be marked in point on the map according to their location, and each transfer descriptor $< lnk >$ marked in line between state-linked resources.

Actually, general HTML elements and contents can be seamlessly integated in between $< loc >$ and $< lnk >$ tag pairs, which may be plain text, images, realtime video streaming etc. When mouse hovering on any $< loc >$ or $< lnk >$

visual element, the inserted HTML will be presented in the pop-up information window, as shown in Figure 4.7.

Figure 4.7: Geo-Visualizer based on Web Map

Detailed information about each descriptor, subordinate attributes and their usages is listed as below (Attributes marked with * are mandatory):

Table 4.1: Resource Descriptor < *loc* > Usage

Name	Type	Value	Usage
		Attribute	
id*	ID	Any valid id defined in XML schema	Uniquely identify a resource in a composition. If left blank, an id generated by system will be assigned
src*	URI		Resource address, usually will be a URL. When extra data channel is needed, it can be expressed as "protocol://ip:portnumber"
name	string	Any valid string not started with the string *data*.	For information purpose
type	enum	"sensor" , "realtime" , "actuator" . Default value is "sensor"	To specify resource type to be sensor, realtime sensor or actuator

Name	Type	Value	Usage
lat	decimal or expression	Valid value from -90 to 90	Resource's latitude. Using user specified expression like *lat* = *data.latitude*, to assign the value of where the key equals to "latitude" from current resource data in realtime manner
lng	decimal or expression	Valid value from -180 to 180	Resource's longitude. Using user specified expression like *lng* = *data.longitude*, to assign the value of where the key equals to "latitude" from current resource data in realtime manner
x	decimal or expression	Valid value from 0 to 100	Resource's relative horizontal position referring to user-defined map. Can be assigned in realtime by expression
y	decimal or expression	Valid value from 0 to 100	Resource's relative vertical position referring to user-defined map. Can be assigned in realtime by expression
		Process	
input	key-value pair or URI		Any assignable input parameters provided by device
filter	expression		Specified by *data.key*. Other data field will be filtered out
viz	enum or URI	Default data visualizer has three options: *area*, *bar* and *line*	For tweaking data visualization effects within information windows. Also external visualization service can be specified by URL
stateml	URI		User-defined StateML file can be specified via URL for execute sequential state transitions regarding single resource
		Style	
style	key-value pair	Options include: *r*, *fill*, *stroke*, *width*	For tweaking resource mark appearance on maps similar to css style. Controllable parameters include radius, filling color, stroke color and stroke width.

Table 4.2: Transfer Descriptor $< lnk >$ Usage

Name	Type	Value	Usage
		Attribute	
id*	ID	Any valid id defined in XML schema	Uniquely identify a state transition chain in a composition. If left blank, an id generated by system will be assigned
src	URI		The address of uploaded csv file can be assigned for bach processing state transition chains.
name	string		Currently only for information purpose
type	string		Reserved filed
points	expression	Should be multiple loc ids that separated by token ";"	To visually line up different loc marks on maps
		Process	
stateml	URI		User-defined StateML file can be specified via URL for establish state tranfer chains involving multiple resources
function	expression or URI	Inlined methods including: $THRESHOLD$, $CEILING$, IF, $LINEAR$	Provide simple intermediary functions for mapping the states of input and ouput nodes. Complicated processing can be done by specified external service by URL
		Style	
style	key-value pair	Options include: *type*, *fill*, *stroke*, *width*	For tweaking link appearance between resources similar to css style. Controllable parameters include line type (default value is *solid*. Optional is *dotted*), filling color, stroke color and stroke width.

4.3.2 HSML Usage Sample

In the following part, we will introduce some typical usages of HSML.

First sample introduces a soil humidity sensor using resource descriptor with id "sesnsorSample". In single composition task, each pair of $< loc >$ tags must

be assigned with a unique id to distinguish each other, otherwise only the first pair $< loc >$ with same id will be regarded valid resource. If left blank, system-generated id will be assigned by default. Though $< lnk >$ tags are processed independently from $< loc >$, still we don't recommend to share same id between $< loc >$ and $< lnk >$.

```
<loc id="sensorSample" name="Soil humidity sensor at Mita"
  input="samplingPeroid: 3" lat="35.6499948" lng="139.7433191"
  x="10" y="20" src="http://www.resource.com/sensor" type="
  realtime" viz="bar" filter="humidity, timestamp" style="r:1;
  fill: red; stroke:black; width:0.2;"></loc>
```

Attributes *lat* and *lng*, as well as x and y must be used in pairs. The former can be mapped to any map interfaces based on global geographic coordinate system, while the latter works for user-defined maps (e.g. indoor maps). Taking the top-left corner as origin of coordinates, $X =$ "10", $y =$ "20" denotes relative location at 10% width and 20% height of the map. At current version, HSML parser will prompt invalid error information unless one pair of location specification attribute presented.

src is a required attribute to specify resource address, which will be directed to the actual resource, which might be a static data repository, an uploaded data file, a virtual sensor, or a dynamic IoT service etc. Together with the attribute *type*, they decided the way how data will be accessed and processed. In the sample, when $type =$ "*realtime*" and $viz =$ "*bar*", it will render realtime data visualization in the information window attached to the resource marker pinned at map geo-visualization interface. If *viz* attribute is not specified, the information window will only show the realtime data in plain text. If *type* is not specified as *realtime* or *actuator*, it will be treated as a static sensor and the previous 20 records will be displayed by default. If the IoT developer leaves $filter$ blank, all the data field within one record will be present or visualized by default.

style attribute has a group of parameters which can manipulate the appearance of each $< loc >$ marker. If left unassigned, default appearance will be a point with 10 pixel radius, with light blue color and 1 pixel red stroke.

Next, we use a drone as our actuator sample. *stateml* specifies a stateML file to control drone to finish a series of predefined commands. While the expressions $lat =$ "*data.latitude*", $lng =$ "*data.longititude*", different from the previous sample, dynamically obtain realtime coordinates data from the associated resource, i.e. the drone, and trigger HSML geo-visualizer to redraw the position of the resource marker on map interface. Thus we can monitor realtime motion and

trajectory of those IoT devices with dynamic position information. It is noteworthy that the suffix following the common prefix *data.* must be mapped to the actual variable names contained in the device data record, e.g. some devices use data fields x and y instead of lat and lng. In this case, corresponding HSML must be $lat =$ "$data.x$", $lng =$ "$data.y$". Expressions in HSML also allows nested variables, e.g. *data.coordinates.lat* and *data.coordinates.lng*.

```
<loc id="actuatorSample" name="Drone" type="actuator" src="
  http://www.resource.com/drone" stateml="uploads/TestUser1/
  1494770866000/predefinedMovement.xml" lat="data.latitude"
  lng="data.longtitude"><img src="http://www.resource.com/
  drone/cameraView/nphMotionJpeg?Resolution=300x240&Quality
  =Standard" style="transform: rotate(180deg);width:305px;">
</loc>
```

Readers may also notice that here we inserted an $< img >$ tags inside HSML, which is the realtime video streaming provided by drone camera. Original HTML tags can be seamlessly blended with HSML thus rich hypermedia contents can be inserted inside into resource composition, including but not limited to plain texts, images, audio, video and etc.

After resources being declared by resource descriptor $< loc >$, they can be further used together with service descriptor in state transfer descriptor $< lnk >$. For example in the following HSML, the previous two resources *sensorSample* and *actuatorSample* are further mapped by a $LINEAR$ function nested in a $THRESHOLD$ function. The $LINEAR$ function is one of the inlined processing functions, which expects three parameters: $LINEAR(y, x, gradient)$. y will be assigned the value of x times $gradient$. If the third parameter is left blank, the gradient of the function will be set to 1 by default. Similarly, function $THRESHOLD$ expects 4 parameters: $THRESHOLD =$ "$y, x, lowerBound, upperBound$", which specifies that the expression y will be executed, when the value of variable x is either exceed the $upperBound$ or drop blow the $lowerBound$.

```
<lnk id="linkSample" name="irrigationTask" function="THRESHOLD
  (LINEAR({actuatorSample.lat, actuatorSample.lng}, {sensor
  Sample.lat, sensorSample.lng}), sensorSample.humidity,
  20, 100)"></lnk>
```

Here the range of relative humidity is from 0% to 100%, which implies only when the sensor data is observed to be less than 20%, its location data will be assigned to the actuator - the drone. This operation will drive the drone to fly to the same position as the sensor.

Last but no least, the final sample showns that how to batch processing large-

scale date using user-uploaded csv files. HSML supports different local data file formats including XML, CSV and JSON etc. In order to better separate resource descriptor groups and transfer descriptor groups, we stipulate to use file name extensions of *.locs* and *lnks* respectively.

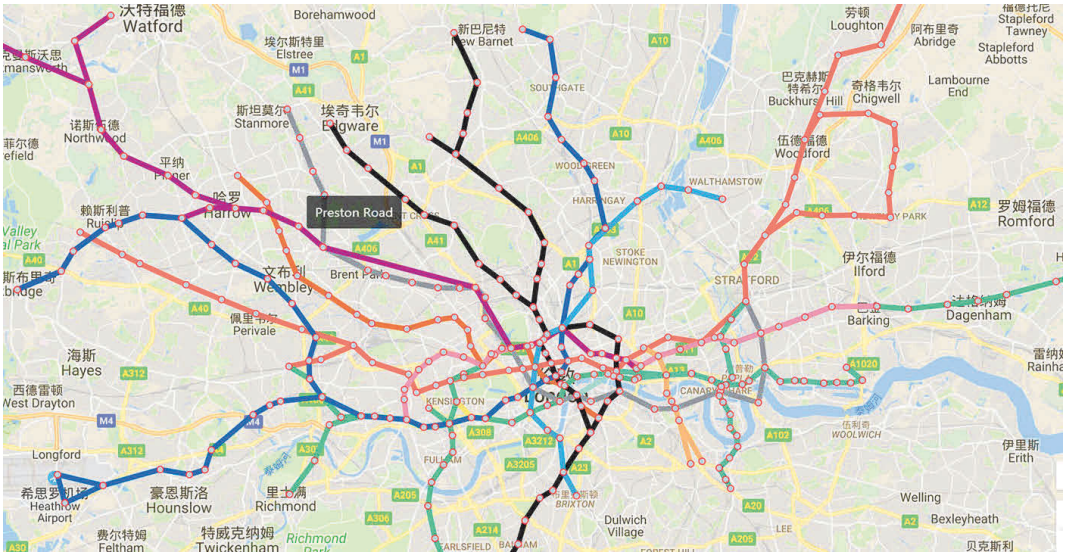


Figure 4.8: London Metro Map by HSML

```
<loc id="metroStation" src="uploads/testUser1/1494806691000/metro
    .locs"></loc>
<lnk id="metroLine" src="uploads/testUser1/1494806691000/metro.
    lnks"></lnk>
```

We used the open data provided by London metrolines. Each record in *metro.locs* stands for a station on a metro line:

```
"id","lat","lng","name","display_name","zone","total_lines"
1,51.5028,-0.2801,"Acton Town","Acton Town",3,2
2,51.5143,-0.0755,"Aldgate",NULL,1,2
3,51.5154,-0.0726,"Aldgate East","Aldgate East",1,2
4,51.5107,-0.013,"All Saints","All Saints",2,1
5,51.5407,-0.2997,"Alperton",NULL,4,1
7,51.5322,-0.1058,"Angel",NULL,1,1
8,51.5653,-0.1353,"Archway",NULL,2.5,1
9,51.6164,-0.1331,"Arnos Grove","Arnos Grove",4,1
10,51.5586,-0.1059,"Arsenal",NULL,2,1
......
```

The content of *metro.lnks* looks like below, when the *points* attribute in *lnks* specifies the line to be drew in between any two adjacent stations.

```
id,points,line,stroke
1,"11,163",1,#AE6017
2,"11,212",1,#AE6017
3,"49,87",1,#AE6017
4,"49,197",1,#AE6017
5,"82,163",1,#AE6017
6,"82,193",1,#AE6017
7,"84,148",1,#AE6017
8,"87,279",1,#AE6017
9,"113,246",1,#AE6017
10,"113,298",1,#AE6017
......
```

The result is shown in 4.8.

4.4 Central Service Orchestration

After IoT developers input and submit their HSML via HSML editor (usually in a web browser) to the composition server, it will be processed by central service orchestration module and eventually interpreted into executable device operations on target devices by host service, as shown in Figure 4.9. Firstly, developers can start with acquiring standard resource descriptions by querying StateML Files registered on the resource server, which are generally supposed to be provided by the service developer. IoT Developers are supposed to explicitly specify resource address, authority information, as well as connection protocol and port number if extra data channel is presented other than HTTP by default.

Submitted HSML texts are parsed by a regular expression interpreter, which is a basic component of central service orchestration. For example: a line of HSML such as $< loc\ id =$ "$mySensor1$" $src =$ "$www.mydomain.com$" $>< /loc >$ will be parsed into standard data format and then stored in a child node of a global object *locs*, like $\{loc : \{id :$ "$mySensor1$", $src :$ "$www.mydomain.com$"$\}\}$. In one composition, *locs* may have one or multiple child nodes, depending on how many *loc* tags there are within the task. The same goes for *lnk* tags.

Parsed data is used to configure corresponding message brokers, according to which each message broker will establish and maintain a linking route table. A linking route table is similar to a routing table that records the senders and receivers of certain state messages. By default, message brokers will then construct

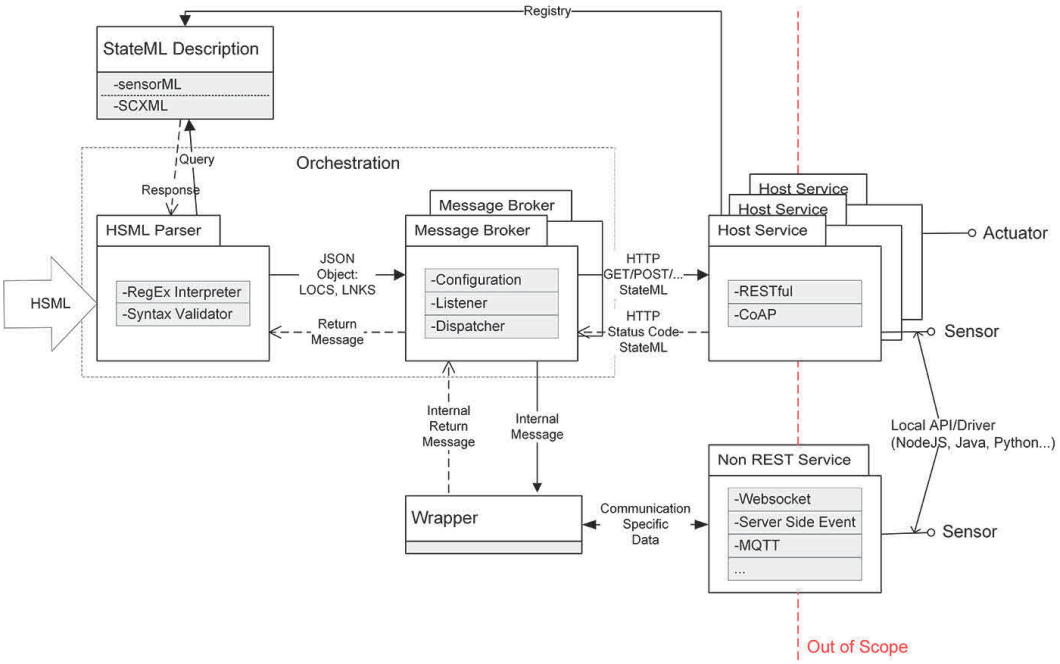


Figure 4.9: The Interpretation Mechanism of HSML

HTTP messages out from a sender's data, and then deliver to receiver's host service according to certain linking rules. As soon as the host service receives the message, it will switch the HTTP operators and parameters, and then call functions accordingly. Following is a simple example written in Node.js. If the web service running at http://www.mydomain.com received a HTTP message with the verb of POST or GET, it will call corresponding functions. There are four HTTP operations used by our system in total: POST, GET, UPDATE and DELETE, which are usually used to trigger specific state transfers of target IoT devices.

Besides the default request/response messaging using HTTP, our framework also supported alternative subscribe/publish messaging models, such as MQTT, server-side event etc. It allows sensor side to spontaneously send data to the message broker. Specifically, extra wrappers need to be explicitly declared, and IoT developers need to specify its protocol, IP address, port number or sometimes data channel etc using HSML, and message broker then will create compatible wrapper instances to establish sub/pub data communications.

```
router.route('/')
.post(function(req, res){...})
```

```
.get(function(req, res){...});
```

Finally, host service will call functions that can be translated to executable codes in local API or drivers, which are usually dependent on some kind of interpreted language, e.g. Java, Python, NodeJS etc. The advantage of interpreted languages is that they provide certain kind of virtual machines that can generate target machine code at runtime to guarantee "write once, run anywhere" without compiling stage (which is required by the compiled language like C). In case of some target devices that are too resource-constraint to host any VM (for example Arduino), a VM hosted device (for example: raspberry pi) can be used as a gateway to communicate with target devices. In these cases, the specific communication methods between gateway devices (e.g the raspberry pi) and target devices (e.g. the Arduino) are still needed to be developed, to support the platform-agnostic feature of the whole framework.

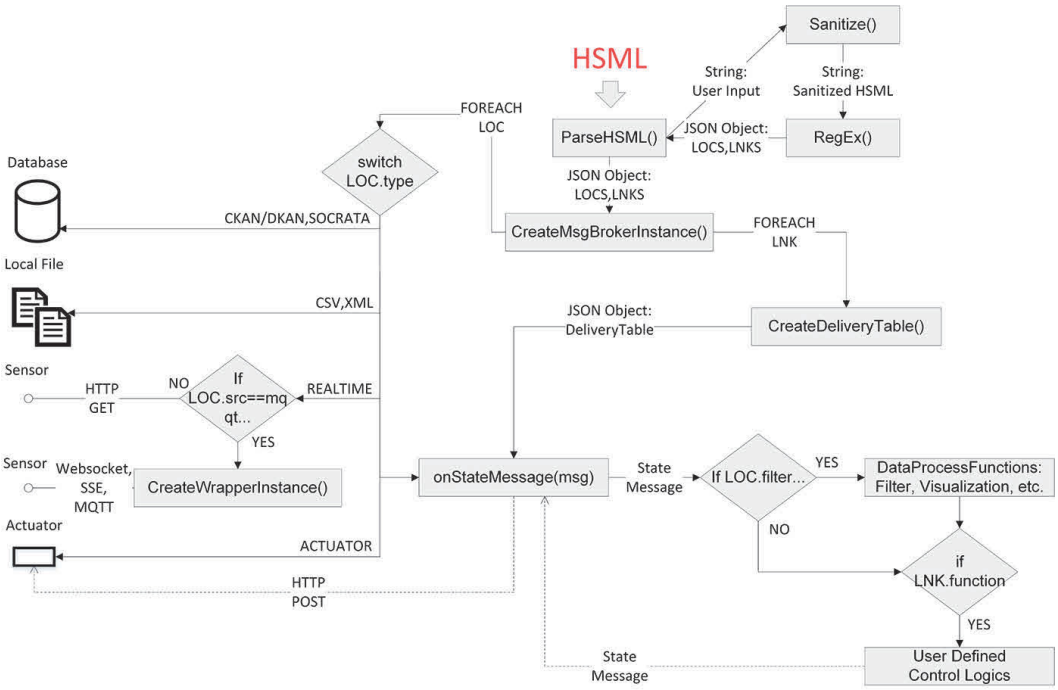


Figure 4.10: Function Flow Diagram

The whole procedure above of service orchestration was actually implemented as the following function flows, as shown in Figure 4.10. When the submission of HSML string triggered the function $ParseHSML()$, it first will pass user-input

string object to *Sanitize*() to detect if there exists any invalid HSML syntax. Elements and attributes that are not listed in white list will be screened out and hence ignored. Sanitized HSML then will be parsed into JSON object *locs* and *lnks*, which respectively contains all *loc*/*lnk* tag information, e.g. their attributes and values.

After parser module, message broker module will initiate at least one single message broker instance according to received *locs* and *lnks* objects. For each *loc* in *locs*, the default message broker will check user-specified *loc.type* and switch to the resource interface accordingly. To specifically mention that, when a resource's type is designated as *realtime* sensors, the message broker instance will request data from the address that specfied by *loc.src* via standard HTTP operations and 80 port by default. While it is possible to establish extra realtime streaming channels by adopting protocol-specific wrappers.

Currently, HSML allows IoT developers to specify the usage of wrappers in the way like "PROTOCOL://URL:PORT" in *loc.src*, e.g. `mqtt://www.example.com/sensor:1883`. In this case, HTTP messages that convey related resource state information, which possibly includes complete data schema that helps annotate raw data, are separated from the I/O data channel that may return streams like "7248,26.3 7248,26.4 7250,26.6 7251,28.3..." Together they constitute a meaningful data resource. Sometimes when stream workloads become too heavy, message broker deployed on cloud servers are able to "replicate" itself and take over communication-intensive task, by leveraging the elastic computing power of the clouds.

Another important function of message broker is to construct linking route table for state messaging by analyzing each *lnk* in object *lnks*. Each record in the delivery table consists of three variables, $msgSource$, $msgTarget$ and $msgOption$, e.g. {192.0.2.163, 192.0.2.144, {option}}. $msgOption$ can be used for describing connection information like extra data streaming channel etc. If one source node is supposed to send state messages to multiple target nodes, it will be merged into single record for management purpose.

The delivery table object will then be passed to the function $onStateMessage()$, which catches state transfer messages from both external nodes and internal components. When certain piece of state message arrived, e.g. "DATA_UPDATED" packed with newly updated data, $onStateMessage()$ will then refer to the delivery table and transfer to whoever expects this message. Sometimes IoT developers order some data processing and/or visualization beforehand, $onStateMessage()$ will

check the attributes $loc.filter$, $loc.viz$ and etc., then send to the proper process modules. Similarly, attribute $lnk.function$ carries user-defined logic that decides the way how one node's state transfer linked to another. A simple example is using IF logic element to indicate when some condition becomes true, an internal state message "CONDITION_TRUE" will be sent to the next node in delivery table to trigger next action. By combining multiple logic elements, IoT developers can realize complicated control task.

Special usages like batch uploading *loc* and *lnk* texts by CSV files, or predefined sequential task by specifying *loc.stateml* attribute, we have not included in this diagram to avoid complexity. Please refer to the following usage samples and appendix for more technical details about HSML and its mechanism.

4.5 Typical Deployment Cases

4.5.1 Two Deployment Patterns

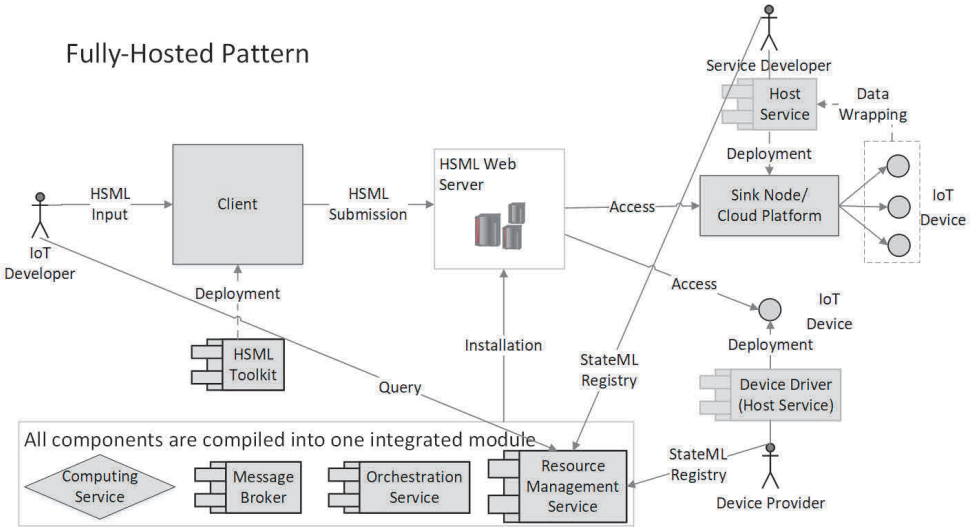


Figure 4.11: A Typical Deployment in Fully-Hosted Pattern

In real practise, the proposed framework can be deployed in **fully-hosted pattern** and **self-hosted pattern**. The full-hosted pattern allows all IoT developers to compose their IoT services the same way as using any other web services. As shown in Figure 4.11, most components including orchestration service, resource management service and extensive wrappers etc., are wrapped up

into an integrated software which is allocated at our HSML web servers. Due to the particularity of message brokers, we have stripped it apart from central service orchestration module. At client end, IoT developers are able to create and edit their HSML texts using the development toolkit running on their web browsers. User-generated HSML texts will later be analyzed and used for configuring message broker instances by central service orchestration. Message brokers and related modules then will take over the actual execution and fetch necessary resource and service data. While third-party service developers or device owners are supposed to register their host service by uploading corresponding service description files to our resource management module. So that IoT developers will be able to query over available IoT services and obtain necessary information for using them.

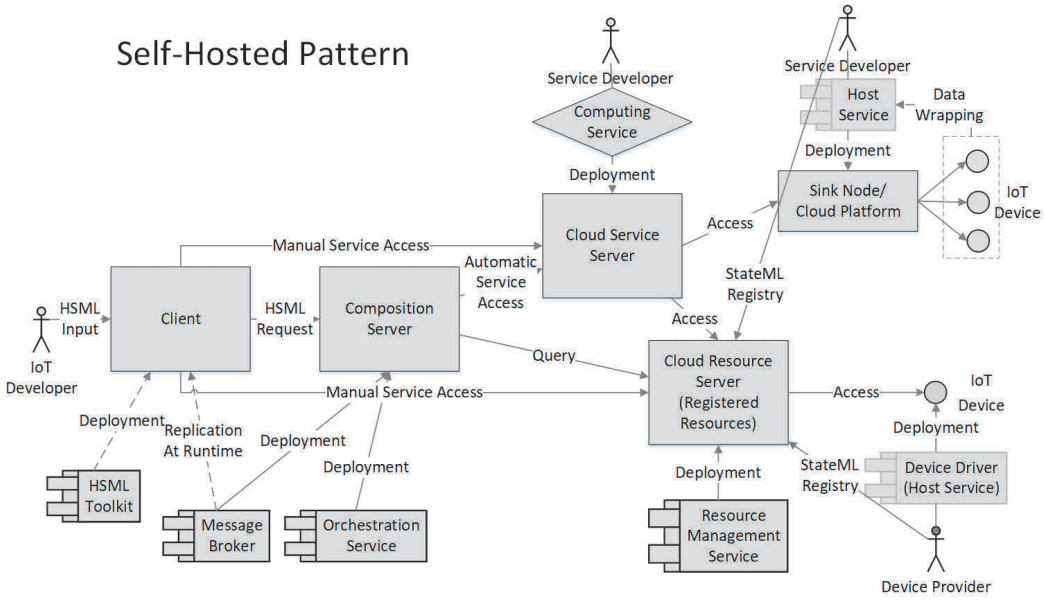


Figure 4.12: A Typical Deployment in Self-Hosted Pattern

In few occasions, service developers, HSML system administrators and IoT developers come from same group of users; But more frequently, IoT and Web services are provided by scattered entities and managed on distributed server clusters. Therefore, self-hosted pattern of deployment allows to download source code package, and deploy each independent framework module in geographically dispersed, cross-organizational environment, as shown in Figure 4.12. It is also possible to tailor system functionality to cover the interests of different stake-

holders. While the central composition server, resource server and service servers stay independently from one another, they are mutually accessible if only they support standard communication protocol and standard Web messaging. If necessary, multi-tier resource server structure can be adopted to further increase the extendibility when sensor nodes scale up or multiple device networks need to be merged together. Self-hosted pattern allows more flexibility and openness in module deployment and customization, and is believed to be particularly applicable to the scenarios such as smart cities, where there is a need to integrate different IoT device networks that deployed and managed by multiple institutions and organizations.

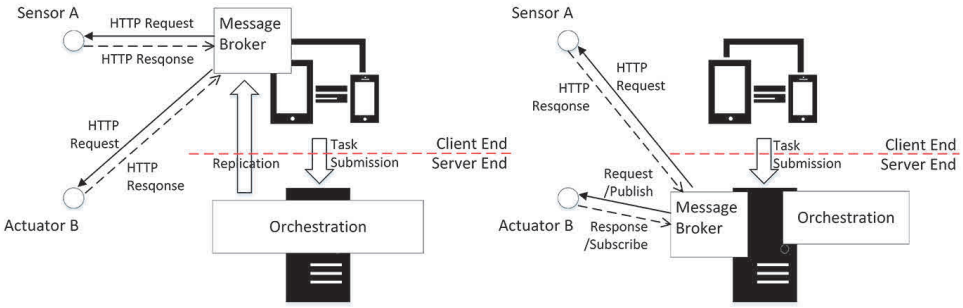


Figure 4.13: Fat Client Model v.s. Thin Client Model

According to different locations that message brokers are deployed, typical implementations can be divided into **fat client model** and **thin client model**, as shown in 4.13. In fat client model, message brokers are replicated on each client end. Therefore there is no need to store client information, meanwhile the computing power of client end can be well exploited. Thin client model is typically devised for relatively small-scale IoT composition tasks developed for multiple end users.

While in thin client model, message broker is deployed together with the central orchestration service that located on the composition server (usually a cloud server). In this case, the central service orchestration, possibly cooperated with external load balancing service, is supposed to elastically allocate sufficient computing resources, and extend the amounts of message broker instances to manage computing-intensive tasks, or assign extra dedicated message broker to take over the real-time communication with nodes of large data flow. Thin client model is devised for those composition tasks with large amount of service nodes involved.

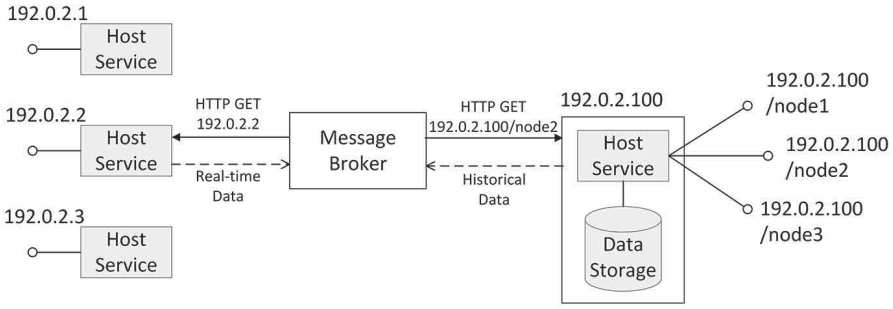


Figure 4.14: Virtual Device Pattern v.s. Realtime Device Pattern

According to different servitization strategies that service developers adopt, it will sometimes affect the availability of composed services. In Figure 4.14, we have simulated two typical patterns. In the situation of local servitization or edge servitization, the message broker retrieves data from device node that locates at 192.0.2.2, it sends a request to the nearest host service to obtain real-time data. In this case, device data usually will not be stored, because of the limited data storage of edge devices. If this is the case, physical device offline may cause composed service unavailable.

While in cloud servitization, device data can be gathered and stored in a central cloud server temporarily or permanently. Instead of sending request to the node itself, the message broker instead retrieved data from service server at 192.0.2.100, and path names (/node5) are used to identify different device nodes. It is worthy of specific attentions that host service will respond with the last updated data, even the actual node may be down. Device offline will not influence the availability of composed service, though the retrieved data may be obsolete.

4.5.2 Deployment Case I: Environment Monitoring

Environment monitoring is among the most widely deployed IoT application, for instances in smart city and smart home areas. In those deployment scenarios, providers have to survey IoT products from different vendors, select proper hardware devices, and integrate heterogeneous subsystems according to target environment and requirements. IoT cloud platforms provide run-time elasticity, unlimited storage and computing capability for this kind large-scale, cross-domain applications. Still, these IoT services require to be accessed, monitored, and ma-

nipulated in a unified manner with central orchestration, and can be composed or decomposed in response to different stakeholders' interests.

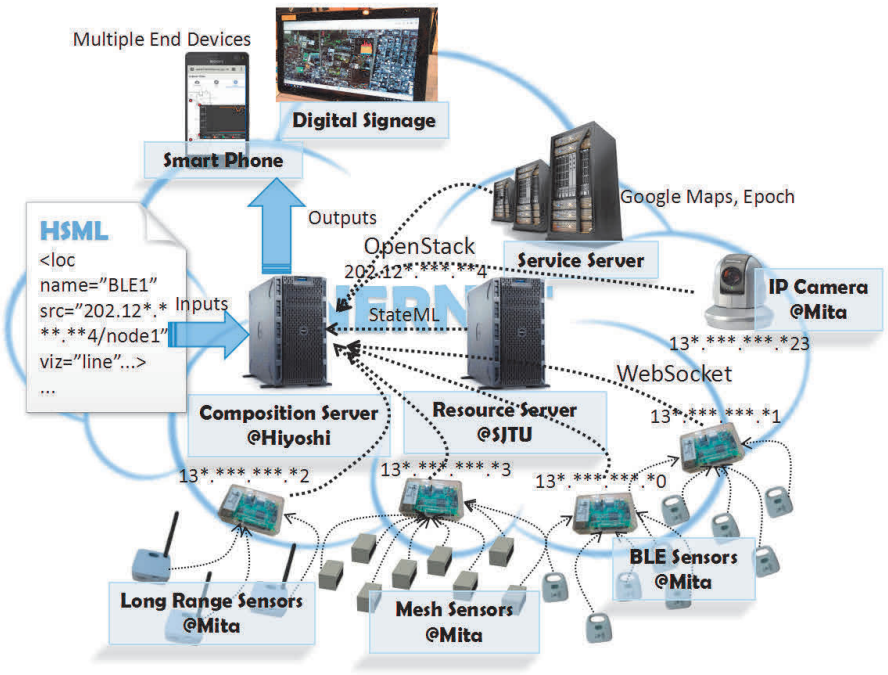


Figure 4.15: Deployment Layout of Case I

In the first deployment case, there were three different types of wireless sensor networks being deployed in and around university campus to monitor the environmental information. Integrated smart sensors were adopted to detect humidity, acoustic noise, temperature, illuminance and pressure. And collected data were visualized in real-time manner. The results can either be viewed on any smart devices by accessing specic website address or from the digital signage at the spot.

In consideration of security issue, part of the real network IP addressed used in case I and II have been concealed. Actual HSML texts used for establishing Case I composition are listed as below:

```
<loc x="48.5" y="26.5" id="mesh1" src="http://202.12*.***.**4/
    node1" type="realtime" viz="line" filter="humi, light,
  temp, sound"></loc>
<loc x="44" y="20" id="mesh2" src="http://202.12*.***.**4/node2"
    type="realtime" filter="humi, light, temp, sound,
  press"></loc>
```

```
<loc x="37" y="11" id="mesh3" src="http://202.12*.***.**4/node3"
    type="realtime" filter="humi, light, temp, sound,
  pres"></loc>
<loc x="32.5" y="21.5" id="mesh4" src="http://202.12*.***.**4/
    node4" type="realtime" filter="humi, light, temp,
  sound, press"></loc>
<loc x="48.5" y="70.5" id="mesh5" src="http://202.12*.***.**4/
    node5" type="realtime" filter="humi, light, temp,
  sound, press"></loc>
<loc x="44" y="77.5" id="mesh6" src="http://202.12*.***.**4/node6
    " type="realtime" filter="humi, light, temp, sound,
  press"></loc>
<loc x="37" y="86" id="mesh7" src="http://202.12*.***.**4/node7"
    type="realtime" filter="humi, light, temp, sound,
  press"></loc>
<loc x="32.5" y="75" id="mesh8" src="http://202.12*.***.**4/node8
    " type="realtime" filter="humi, light, temp, sound,
  press"></loc>
<loc x="78.8" y="23" id="ble1" src="http://202.12*.***.**4/node9"
     type="realtime" viz="line" filter="humi, light,
  temp, sound"></loc>
<loc x="78.8" y="39" id="ble2" src="http://202.12*.***.**4/node10
    " type="realtime" filter="humi, light, temp, sound,
  press"></loc>
<loc x="93.5" y="23" id="ble3" src="http://202.12*.***.**4/node11
    " type="realtime" filter="humi, light, temp, sound,
  press"></loc>
<loc x="93.5" y="39" id="ble4" src="http://202.12*.***.**4/node12
    " type="realtime" filter="humi, light, temp, sound,
  press"></loc>
<loc x="78.8" y="55" id="ble7" src="http://202.12*.***.**4/node13
    " type="realtime" filter="humi, light, temp, sound,
  press"></loc>
<loc x="93.5" y="55" id="ble8" src="http://202.12*.***.**4/node14
    " type="realtime" filter="humi, light, temp, sound,
  press"></loc>
<loc x="78.8" y="71" id="ble9" src="http://202.12*.***.**4/node15
    " type="realtime" filter="humi, light, temp, sound,
  press"></loc>
<loc x="93.5" y="71" id="ble10" src="http://202.12*.***.**4/
    node16" type="realtime" viz="line" filter="humi, light,
  temp, sound, press"></loc>
<loc lat="35.6499948" lng="139.7433191" id="LoRa1" name="Keio
  Mita Campus North Building" src="http://202.12*.***.**4/node17"
```

```
      type="realtime" viz="area" filter="humi, light, temp,
  sound"></loc>
<loc lat="35.6483988" lng="139.7431656" id="LoRa2" name="Keio
  Mita Campus South Building" src="http://202.12*.***.**4/node18"
      type="realtime" viz="line" filter="humi, light, temp,
  sound"></loc>
<loc lat="35.649226" lng="139.742004" id="LoRa3" name="Keio
  Mita Campus West Building" src="http://202.12*.***.**4/node19"
     type="realtime" filter="seqNum, humi, light, temp,
  sound, press"></loc>
```

Each $< loc >$ described a related sensor resource, including 10 BLE type sensors, 8 mesh network sensors as well as 3 long range sensors. Corresponding host services were launched in three sink nodes respectively, i.e. the Raspberry Pis. Service description files in StateML were registered at the Cloud server based on OpenStack with IP address 202.12*.***.**4, which physically located at Shanghai Jiao Tong University, Minhang Campus. On composition server, HSML accessed sensor host services via URLs, and extra data streaming channels were established using WebSocket wrappers.

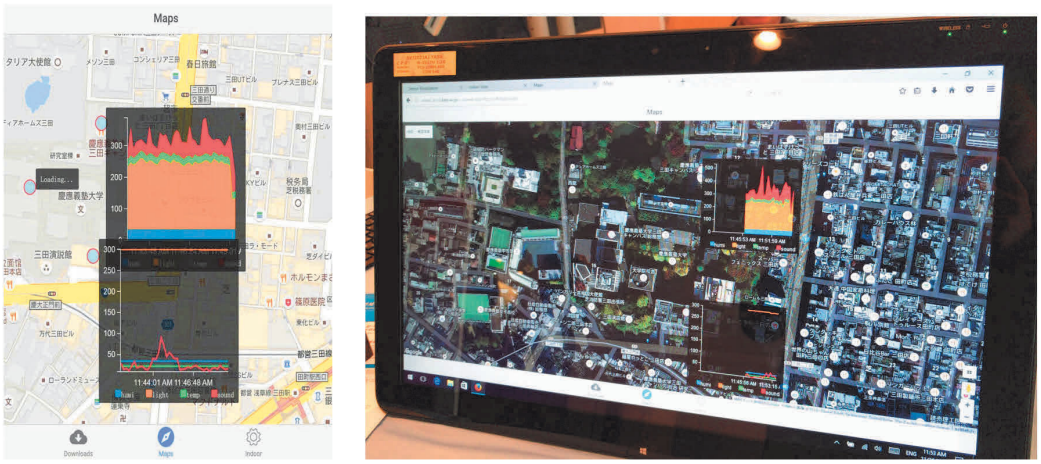


Figure 4.16: Composition Result of Case I on Mobile (left) and Digital Signage(right)

Meanwhile, we also used HSML to integrate necessary web service components of Angular Google Maps, which can be regarded as a web service version of Google Maps API, and Epoch for real-time data visualization. Sensor resource was first implicitly composed to the visualization component by specifying the attribute *viz* in each $< loc >$ tag, then all the visual information will be further assembled to

a Google map canvas according to designated location information of each sensor node, as shown in Figure 4.16:

4.5.3 Deployment Case II: Open Automation

Case II presented another typical application scenario of open automation systems. We adopted a different deployment approach from Case I, by shifting part of computing tasks from central cloud servers to edge devices nearer to the leaf device node. As the scale and number of end devices escalate, this kind of decentralized computing architecture prevent servers from being consulted for every little minor detail. Smaller time-sensitive computational decisions can be made by an intermediary device, like a mobile phone or a smart gateway, which then aggregates all the data it learns and upload to the servers [105].

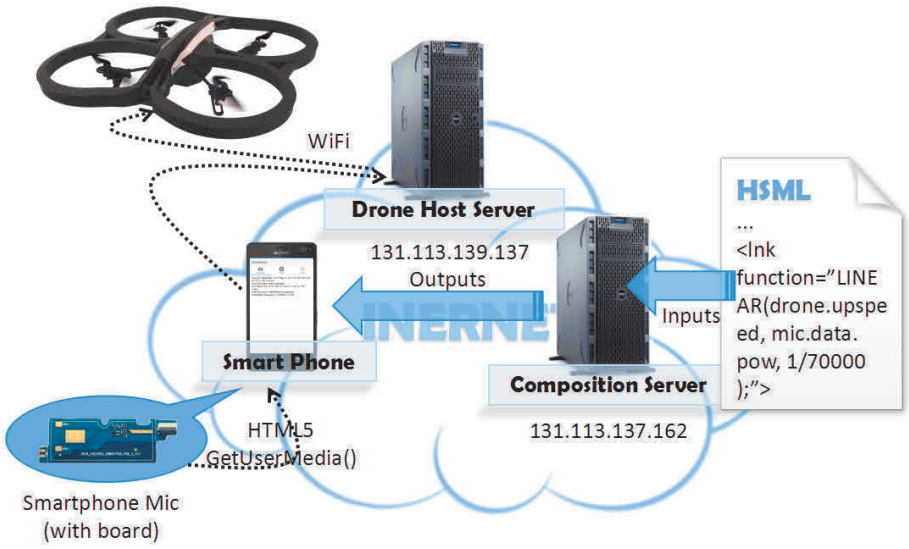


Figure 4.17: Deployment Layout of Case II

We managed to control a drone's lifting speed by blowing air into a mobile phone's mic, which, in its nature, established a state transfer chain between mobile mics white noise intensity (sensor's state) and drones lift speed (actuator's state) and applying a linear linking rule to it. The mobile phone's white noise signal was processed by a local host service (JavaScript mainly) running on the web browser, and calculated results were directly obtained by the message broker,

which was duplicated from central service orchestration to be running on the same web browser. Similarly, the drone's host service (NodeJS) located in a server. It translated related state transition messages sent by message broker to actual control commands, and sent to the drone via Wi-Fi signal. For brevity, resource server was omitted here. HSML texts for Case II are listed as below:

```
<loc id="mobileMic" src="http://localhost/mic" x="15"
  y="30"></loc>
<loc id="drone" type="actuator" src="http://13*.***.***.**7:
  1337" x="15" y="40" stateML="uploads/testUser1/1494770866000/
  initial.xml"></loc>
<lnk id="flappyBirds" function="LINEAR(drone.upspeed,
  mobileMic.data.pow, 1/70000);"></lnk>
```

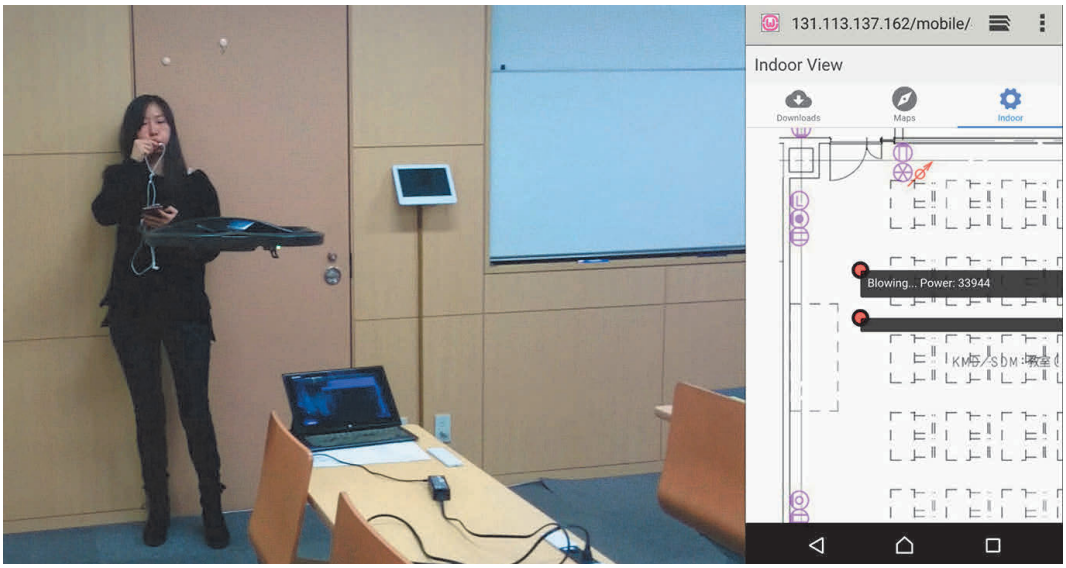


Figure 4.18: Composition Result of Case II

In Case II, HSML was much simpler than Case I, since there are only two resources declared. As the *stateml* attribute in the second $< loc >$ specified an uploaded XML file. This file initialized drone's default behavior of taking off, hovering at 2 meters high, and then flying forward and down at the speed of 1 m/s and 0.2 m/s respectively. We have already introduced how to use stateML description to assign a sequential state transfer task in section 3.5.2, and the content actually looked like this:

```
<state id="takingOff">
    <transition event="height: 2" target="hovering">
```

```
    </transition>
</state>
<state id="hovering">
    <transition event="forwardSpeed: 1; downSpeed: 0.2"
    target="flying"></transition>
</state>
```

The composition server parsed related HSML and acquired the sensor and actuator that declared by each resource descriptors by standard HTTP method GET. The state-transfer descriptor $< lnk >$ assigned the value of mobile mic's white noise intensity to the drone's lifting speed at the gradient of 1: 70000.

4.6 Summary

Based on the concepts of state and state transfer introduced in the previous chapter, this chapter presented an open IoT service composition framework, specified each core components within the framework and provided a typical implementation of the overall framework.

As the prerequisite of proposed framework, each and every IoT device must be encapsulated into URI-deferencable web resource with general state-based interfaces. The host services of IoT devices were required to provide seamless interoperability and open accessibility of sensor/actuator/IoT node from different vendors via uniform management, as an intermediate abstraction layer separating atop application logic from of-bottom device APIs/drivers. According to different locations that host services were launched, mainstream servitization could be divided into three categories: local servitization, edge-based servitization and cloud-based servitization.

A complete proposed framework consisted of a few core modules, including: 1) A domain-specific-language based web development toolkit with HTML-like syntax, namely Hyper Sensor Markup Language (HSML); 2) A state-transfer-based central service orchestration as the service composition mechanism; and though not strictly a module, 3) A machine-readable unified resource representation (stateML) specifically for describing FSM-modelled IoT service interfaces, as already introduced in the previous chapter. There were also other modules like resource manager in charge of IoT service registry and query, as well as extensible wrappers to support optional data streaming channels like WebSocket, Server-side event, MQTT etc.

In Section 4.3, we introduced in details the web application development toolkit, HSML, which provided three kinds of descriptor: resource descriptor, service descriptor and state-transfer descriptor. It allowed IoT developers at all levels to describe various IoT service nodes and the interrelations between them in a concise and flexible way. HSML syntax, graphic user interface as well as typical usage sample were also explained.

Followed by the underlying central service orchestration mechanism based on state transfer. It was in charge of the interpretation of user-generated HSML texts, coordinating and composing IoT service nodes into customizable and value-added applications. The central service orchestration comprised two key components: HSML parser and message brokers. While the parser analyzed HSML texts and generated linking route tables, the message broker instances were supposed to actually access service node, establish optional data streaming channel using wrappers if necessary, manage state transfer flow between sequential service nodes according to predefined linking route table.

To live up with different deployment needs, proposed framework provided 1) Fully-hosted deployment pattern and 2) Self-hosted deployment pattern. The former allowed IoT developers to use our all-in-one IoT service composition platform the same way as using any other websites. While the latter allowed source code package download to fully customize their own composition platforms. We also discussed a few more alternatives when there is a need to deploy some system modules in distributed computing environment.

Finally, two actual deployment cases were presented to show the feasibility of deploying proposed framework under distributed service architecture. The first case of environment monitoring was a typical example for wide-range, large-scale and cross-organization IoT applications. While second case of open automation showed that proposed framework was also able to handle highly customizable applications with transitional task logic.

Chapter 5
Evaluation

In the evaluation, we carried out a systematic assessment of proposed framework to evaluate whether the aforementioned four research issues were improved and to what extent. As shown in Fig. 5.1, our evaluation strategy was comprised of 1) user test, 2) expert interview and 3) architectural comparison.

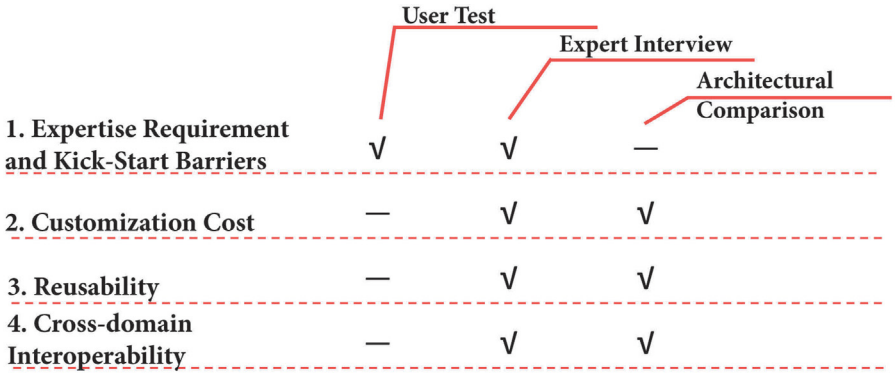


Figure 5.1: Evaluation Strategy

Since expertise requirement and kick-start barriers were compare items closely related to user experience, we hence conducted a series of beginner-centered user experiments to test the learnability, sociability, retrievability and task load of our web development toolkit explicitly. A complementary expert interview was also adopted to gain feed backs from veteran IoT developers inside the industry. On the other hand, customization cost, reusability and cross-domain interoperability, were more structural aspects, an architectural comparison together with expert interview were made to systematically review the proposed framework.

5.1 User Test

In this section, the evaluation is focused on the usability of web development toolkit which is codetermined by directly the learning cost of the toolkit itself as well as indirectly to what extent the underlying composition architecture can relieve the kick-start hurdle. Tests on learnability, sociability, retrievability, task load comparisons with selected parallel technologies were carried out, hopefully presenting a relatively comprehensive perspective on how this intermediary framework retrench the (re-)adaptive efforts and entire development cost of IoT applications.

5.1.1 Learnability

In learnability evaluation parts, 20 students were recruited as participants for the evaluation experiment. In consideration of test validity, participants were divided into 3 groups according to different HTML programming experience: no experience, less than 1 year experience, and more than 1 year experience. All participants were the first time to touch on HSML and proposed framework before. Participants were required to learn the functionality of proposed web development toolkit by watching a tutorial video. A brochure was handed out to explain the usage of HSML. The development toolkit was running on web browser on a PC client with Internet connection. Students may watch the video as long as they want and test HSML on the PC. After the initial learning stage, students were required to execute a relatively easy task to test first time performance. The task included 3 steps: 1) Retrieving data: Use HSML to load resources from given URLs; 2) Editing sensor information: Add extra description to resource descriptors, such as: location, name and label; 3) Visualizing: Tweak the visualization effect by attaching related service component attributes. Time consumption (measurement: minute) of each step was recorded. Mean time are shown in the Figure 5.2.

The mean time of learning process has statistical relation with experience (p=-0.8, a⟨0.01), which means the previous HTML experience do help to learning process. However, from the actual value, we can tell that participants with no HTML experience only spend 50% more time than experienced participants. This relatively small difference suggests that there exist no barriers for non-experienced participants for learning.

With the help of reference brochure, all participants successfully completed all 3 steps of the task. Results showed previous HTML experience could significantly

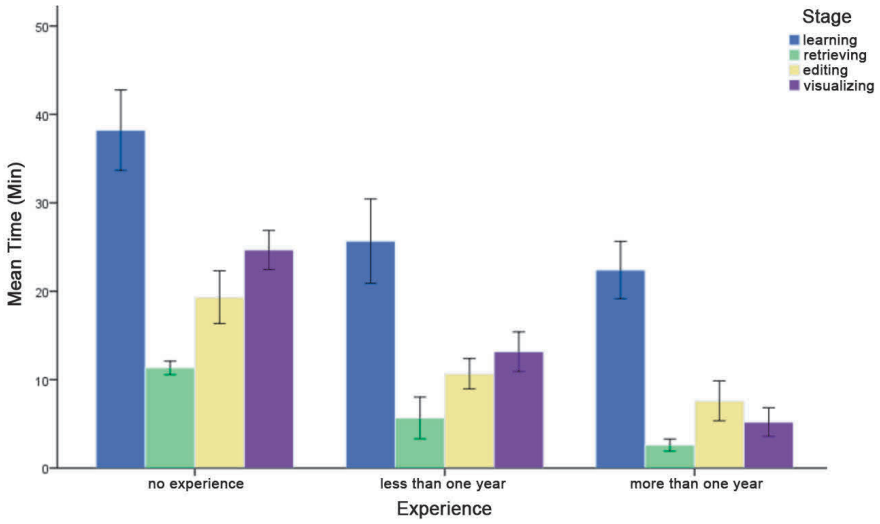


Figure 5.2: Learnability Test Results

improve the first-time performance. And we found that experienced participants felt more encouraged and willing to try out with different attributes and visualization effects using HSML. Besides, even for non-experienced participants, the performances were acceptable. In the interview after the test, most participants, regardless of previous HTML experience, agreed that the proposed development toolkit was easy to use and very effective.

5.1.2 Sociability

In sociability test, 10 students with best performance in learnability test were invited to design sensor data visualization based on environmental sensors on campus. 100 sets of environmental sensor data, including PM2.5, temperature, humidity and noise level, were provided. Information including sensor name, position, source URL and service descriptions provided by their owners. Before the test, students were asked about the willingness of sharing raw sensor data to their friends if they were the owner. After the test, a URL linked to user-generated contents (mainly data visualization and inserted multimedia contents) was created. The previous willingness survey were carried out again. Students willingness of sharing were rated as 5-point LIKERT scale: absolutely not (-2), possibly not (-1), not decided (0), possibly yes (1), absolutely (2).

Figure 5.3 shown that students were more willing to share self-defined visual-

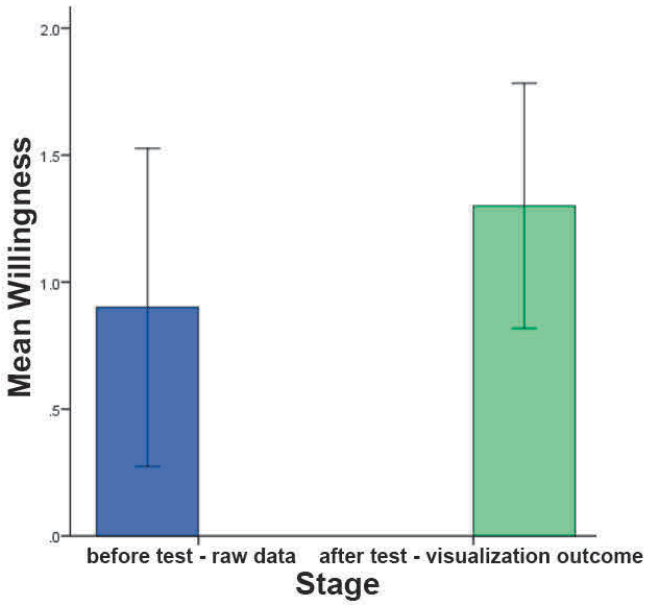


Figure 5.3: Sociability Test Results

ization outcomes than raw data. The difference was statistically significant (sig = 0.021). In the interview after experiment, we asked participants for the reasons why they decided to share or not. Answers showed that the motivational factors that involved in sharing decision mainly included usefulness for others, sense of accomplishment, and sense of pleasure. These outcomes indicated that, the composed outcomes were more likely to be shared because they are more useful to others, or able to bring more accomplishment and pleasure to its developers.

5.1.3 Retrievability

In retrievability test, we evaluated how created HSML might affect the discovery process of the IoT resources in our system. To retrieve specific resources, a query command usually consists of one or more keywords. If an IoT resources information is labeled adequately and properly, it can be easily discovered by matching keywords and labels. However, this ideal situation rarely happens. Most IoT devices in open cloud platforms, usually lack of necessary descriptions to provide clear clues for searching. HSML may help to address this problem from two aspects: First, the HTML syntax itself has already contained certain level of semantic annotation, which will naturally derive necessary labels to describe

sensors; Secondly, analyzing state transfer chains generated by users will help to establish correlations between resources, thus possibly contributing new searching methods that based on linked resources.

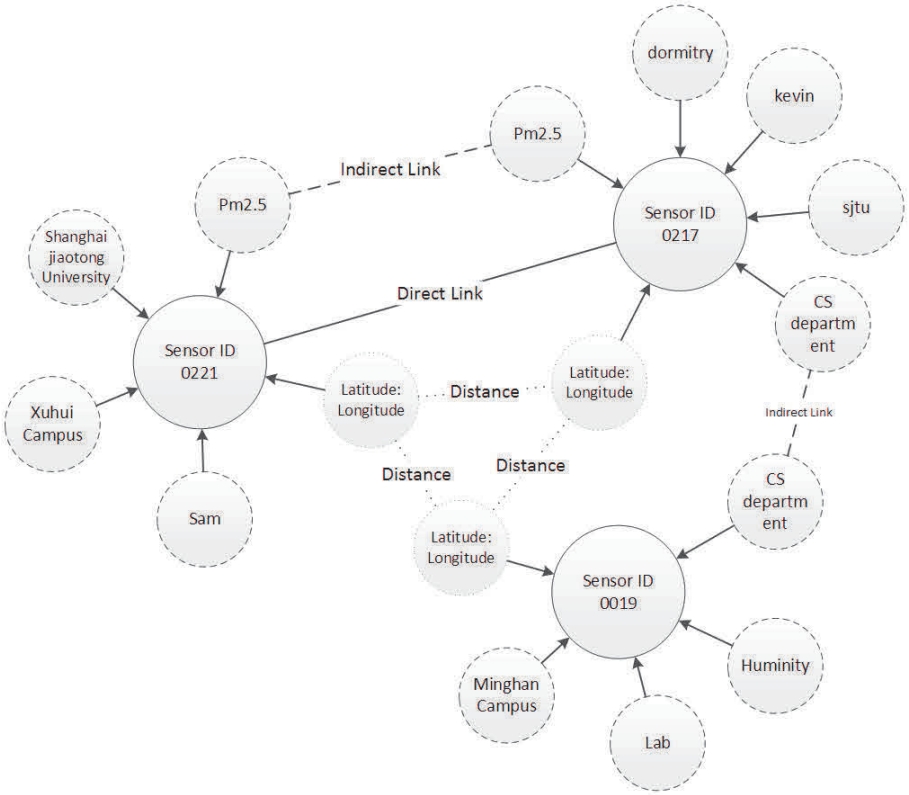


Figure 5.4: Three Types of Relation Information Provided by HSML

To evaluate these effects, we carried out an experiment to compare the performance of an experimental group that used link based searching method and a control group that used keyword matching method. As shown in Figure 5.4, three types of correlations provided by HSML are analyzed: 1) direct link, 2) indirect link, and 3) geospatial distance. They were used to compute the overall link strength between nodes. Direct links were those links defined explicitly in state transfer descriptors, for example: $< lnkpoints = sensor0321, sensor0217 >< /lnk >$ defined a direct link between sensor0321 and sensor0217. The strength of direct link, denoted as S_{DL}, was decided by the frequency and category of $< lnk >$ shown in user-generated HSML text: $S_{DL} = \sum Freq_{lnk} \times Coef_{category}$. Indirect links were those links defined implicitly by semantic labels contained in

user-generated HSML, such as: name attributes. For example, sensor0019 and sensor0217 both had the same label CS department, which would generate an indirect link between them. The strength of indirect link, denoted as S_{IL}, was decided by the count of labels: $S_{IL} = \sum Freq_{label}$. Distance meant great-circle distance between two nodes geographical coordinates defined by the Spherical Law of Cosines: $D = acos(sin\varphi_1 \times sin\varphi_2 + cos\varphi_1 \times cos\varphi_2 \times cos\Delta\lambda) \times R$, where φ was latitude, λ was longitude, R was earths radius (mean radius is 6,371km).

Figure 5.5: Comparison on Retrievability Results

After calculating link strength between each two nodes, a weighted graph

contained all the sensors (as nodes) and their relations (as edges) was constructed and used for link based searching in experimental group. We simulated a set of search queries, for example: PM2.5 Minhang campus dorm, executed them in both experimental group and control group. Results showed that the experimental group provided less failure rate (3%) vs control group (26%), and returned 87% more valid records in average. An example of comparing result from control group (top) and experimental group (down) are shown in the Figure 5.5. Records marked in green rectangle are valid records that were missed in control group. The experimental group also generated a slightly more invalid records (marked in red rectangle). However, since we can use the degrees and edge weights in relation graph to sort the records, the invalid records could be easily excluded by users because they usually have less degrees and edge weights.

5.1.4 Task Load Comparison

The purpose of task load comparison was to reveal how heavy the work load that development tools placed on the IoT developers, specifically beginners without programming experiences. We picked up three representative open frameworks as our compare objects in IoT service composition area, which that adopted different composition approaches respectively. They were: Home Assistant, Node-Red and PubNub Eon, whose detailed mechanisms have been introduced in Section 2.5 previously. A horizontal comparison was made between HSML and the three parallel tools using same tasks.

In the task load test, four participants (2 males, 2 females, age ranging from 20 to 26) were invited to accomplish two basic tasks using four different development tools respectively. The tasks were as follows:

1. Introduce a temperature sensor that connected to an Arduino board into target system and show the data reading.

2. Use sensor data to trigger an actuator, i.e. a LED in this case, to complete an automation. (In PubNub Eon's case, this task was replaced by tweaking data visualization effect as it didn't support automation.)

To avoid bias as possible, we focused on participants who were beginners to IoT development and without programming experiences or technology background. And we also adopted a Latin Square to decide the sequence of all four frameworks

Table 5.1: Test Sequence for Each Participant in Latin Square

Participant	System Sequence			
A	1	2	3	4
B	4	3	2	1
C	2	1	4	3
D	3	4	1	2

to be tested, among which 1 stands for Home Assistant, 2 for Node-Red, 3 for PubNub Eon and 4 for HSML.

All four tools were set up based on same experiment environment, the hardware of which consisted of: 1) An MSI GS60 notebook (with Intel i7-6700HQ CPU @ 2.60GHz, RAM 16.0G); 2) An Arduino Uno board (connected to 1 via COM3); 3) An LM35 Temperature Sensor (connected to 2 via Analog pin 0) and 4) An LED (connected to 2 via Digital pin 11). Operating system was Windows 10 (64bit), and other software that used in the experiment included:1) Web Browser: Chrome; 2) Code Editor: Brackets and 3) Local Server Environment: XAMPP.

At the beginning of the test, each participant was given a paper materials that introduced detailed experiment procedure. During each test, each participant was allowed to raise questions whenever they felt stuck in the task. The question times were recorded, and the time that a participant spent on task 1 and 2 respectively were also recorded.

NASA Task Load Index (NASA-TLX) was adopted in the experiment to evaluate workload that participant subjectively perceived. After each tool was tested, the participant was asked to filled out a computerized NASA-TLX rating scales. We also carried out an unstructured interview for each participant, asking about their preference about the four tools and reasons.

For the first task, the mean time consumption and standard deviation was shown in Table 5.2, while a smaller standard deviation indicates a more convergent rating among all four participants. Paired Sample T-Test(2-tailed) was conducted

Table 5.2: Mean Time Consumption for the First Task

	Tool 1	Tool 2	Tool 3	Tool 4
Mean(s)	1103	425	1155	448
N	4	4	4	4
Std. Deviation	346	56	244	136

to verify statistical significance of the result. When *sig* was lower than 0.05, we basically consider that the possibility for paired two mean values to be equal was less than 8%, which implies a significantly difference:

Tool 1 vs Tool 4: not significant (p-value = 0.72)

Tool 2 vs Tool 4: not significant (p-value = 0.799)

Tool 3 vs Tool 4: significant (p-value = 0.034)

For the second task, due to PubNub Eon was not able to accomplish automation task, we had to rule No.3 result out here. Paired Sample T-Test results were

Table 5.3: Mean Time Consumption for the Second Task

	Tool 1	Tool 2	Tool 4
Mean(s)	948	389	652
N	4	4	4
Std. Deviation	353	70	338

as follows:

Too 1 vs Tool 4: not significant (p-value = 0.112)

Tool 2 vs Tool 4: not significant (p-value = 0.273)

We could draw a conclusion from above that Tool 2 and 4 consumed significantly less time in both tasks. Though Tool 2 seems a litter faster than 4, no significance is found by t-test. Tool 2 and 4 are obviously faster than 1 and 3 in task 1, proved by t-test in 95% confidence interval.

Next we compared all four tools by the mean times that participants raised questions during the overall tasks in 5.4:

Table 5.4: Mean Question Times

	Tool 1	Tool 2	Tool 3	Tool 4
Mean	6	2	6	3
N	4	4	4	4
Std. Deviation	3	0	1	1

Paired Sample-T Test results were as follows:

Tool 1 vs Tool 4: significant (p-value = 0.061)

Tool 2 vs Tool 4: not significant (p-value = 0.495)

Tool 3 vs Tool 4: significant (p-value = 0.05)

The results shown that participants have significantly less questions when using Tool 4 than using Tool 1 and 3, and no significant difference was found between Tool 2 and 4.

Lastly, we have evaluated the overall NASA-TLX ratings and each subindex similarly, including: mental demand, physical demand, temporal demand, performance, effort and frustration. Paired Sample T-Test results were as follows:

Table 5.5: Mean Overall Ratings of NASA-TLX

	Tool 1	Tool 2	Tool 3	Tool 4
Mean	48.400	22.750	43.000	26.975
N	4	4	4	4
Std. Deviation	18.7220	12.6350	14.3129	14.1170

Tool 1 vs Tool 4: significant (p-value = 0.028)

Tool 2 vs Tool 4: not significant (p-value = 0.641)

Tool 3 vs Tool 4: significant (p-value = 0.021)

We can learn from the result that participants gave significantly higher rating to Tool 2 than to 1 and 3. And no significant difference is found between 2 and 4, proved by t-test in 95% confidence interval. Paired Sample T-Test results as

Table 5.6: Mean Mental Demand

	Tool 1	Tool 2	Tool 3	Tool 4
Mean	42.50	23.75	46.25	21.25
N	4	4	4	4
Std. Deviation	15.546	10.308	13.150	7.500

follows:

Tool 1 vs Tool 4: significant (p-value = 0.077)

Tool 2 vs Tool 4: not significant (p-value = 0.664)

Tool 3 vs Tool 4: significant (p-value = 0.03)

Conclusion can be drawn that participants perceived significantly less mental demand when using Tool 4 than using 1 and 3. No significant difference is found between 2 and 4. Paired Sample T-Test results were as follows:

Tool 1 vs Tool 4: significant (p-value = 0.032)

Tool 2 vs Tool 4: not significant (p-value = 0.252)

Tool 3 vs Tool 4: not significant (p-value = 0.861)

Table 5.7: Mean Physical Demand

	Tool 1	Tool 2	Tool 3	Tool 4
Mean	45.00	18.75	27.50	28.75
N	4	4	4	4
Std. Deviation	23.452	14.361	5.000	16.520

According to the result, participants perceived significantly less physical demand when using tool 2, 3 and 4 than to 1 and no significant difference is found between 2, 3 and 4. This may be caused by that we provided full JavaScript code template when testing with Tool 3, which possibly induced bias. Paired Sample

Table 5.8: Mean Temporal Demand

	Tool 1	Tool 2	Tool 3	Tool 4
Mean	48.75	23.75	40.00	31.25
N	4	4	4	4
Std. Deviation	26.575	13.769	14.142	14.930

T Test results were as follows:

Tool 1 vs Tool 4: not significant (p-value = 0.432)

Tool 2 vs Tool 4: not significant (p-value = 0.576)

Tool 3 vs Tool 4: not significant (p-value = 0.473)

We observed that participants perceived no significant difference regarding temporal demand when using Tool 1, 2, 3 and 4. One possible explanation may be that the basic tasks selected for the test were not complicated enough to reflect the temporal demand in real IoT development cases. Paired Sample T-Test results

Table 5.9: Mean Performance

	Tool 1	Tool 2	Tool 3	Tool 4
Mean	40.00	20.00	33.75	23.75
N	4	4	4	4
Std. Deviation	33.417	19.149	21.360	12.500

were as follows:

Tool 1 vs Tool 4: not significant (p-value = 0.250)

Tool 2 vs Tool 4: not significant (p-value = 0.681)

Tool 3 vs Tool 4: not significant (p-value = 0.382)

We observed that participants perceived no significant difference regarding performance among Tool 1, 2, 3 and 4. Paired Sample T-Test results were as

Table 5.10: Mean Effort

	Tool 1	Tool 2	Tool 3	Tool 4
Mean	50.00	21.25	46.25	31.25
N	4	4	4	4
Std. Deviation	14.142	14.361	17.500	21.747

follows:

Tool 1 vs Tool 4: significant (p-value = 0.022)

Tool 2 vs Tool 4: not significant (p-value = 0.343)

Tool 3 vs Tool 4: not significant (p-value = 0.124)

We observed that participants perceived significantly less effort required when using Tool 4 than using 1. And no significant difference is found among 2, 3 and 4. Paired Sample T-Test results were as follows:

Table 5.11: Mean Frustration

	Tool 1	Tool 2	Tool 3	Tool 4
Mean	51.25	25.00	46.25	28.75
N	4	4	4	4
Std. Deviation	27.801	8.165	15.478	21.747

Tool 1 vs Tool 4: significant (p-value = 0.042)

Tool 2 vs Tool 4: not significant (p-value = 0.761)

Tool 3 vs Tool 4: not significant (p-value = 0.133)

Participants perceived higher frustration when using Tool 4 than using 1, but not significant among 2, 3, and 4, proved by t-test in 95% confidence interval.

As an overall conclusion, Node-Red and HSML were proved to have significantly better performance than Home Assistant and PubNub Eon in: time consumption, Problems, NASA-TLX Overall Rating, Mental Demands and Effort. While Node-Red may have slightly better performance than HSML, but not significant in statistical test.

5.2 Expert Interview

In order to obtain a comprehensive appraisal from novice to veteran IoT developers, we have also interviewed three experienced IoT developers and researchers with varying expertise and backgrounds: The first interviewee was a graduate student majored in computer science whose research was centered on smart transportation and vehicle network; Second interviewee was An IT practitioner who had 5-year IoT development experience, and last interviewee was a college faculty who also had 5-year education experiences of teaching IoT development.

We conducted a structured expert interview that consisted of three sections. The first section included five questions targeting individual technical skills and IoT-related education background, e.g. most familiar toolkit and programming language. The second section included also five questions related to IoT development procedure. In this section, we tried to reveal what factors were considered to be contributing to the overall time and effort consumption in IoT application development, and what general features were valued most in an IoT development framework. In the last section of the interview, we asked specific questions after interviewees trying out the proposed framework, to figure out their appraisal if the proposed framework can improve certain aspects of IoT application development.

All of the three interviewees had the experiences dealing with IoT interoperability issues, e.g. integrating two heterogeneous IoT systems based on different standards, protocols, programming languages etc. In general question section, interviewees expressed different needs for IoT framework features, which were to some extent consistent with their concerns on time and effort consumption. In the case of the first interviewee with one-and-a-half years experiences, he put specific emphasis on reliability and community support of especially open IoT frameworks, as he listed environment building, code porting and using 3rd-party software as labor-consuming factors. While according to the other two interviewees with 5-year experiences, their focuses were more on framework's neutrality (e.g. platform neutrality, programming language neutrality, standard neutrality), flexibility (e.g. changing nodes or task logic during run-time) and reusability (e.g. code reusability).

All of the three interviewees reached a consensus that the proposed framework was able to improve customization, resuability and cross-domain interoperability issues compared to their previously used platforms and tool kits. And they were also willing to use the proposed framework in future IoT development. As the

Table 5.12: Expert Interview Results

	Interviewee 1	Interviewee 2	Interviewee 3
Experiences	1.5 years	5 years	5 years
Domain	Smart Transportation	Data Analysis	IoT Education
Valued Features	Framework Reusability >Community Support >Platform Neutrality	Hardware Independence >Standard Neutrality >Language Neutrality	Flexibility >Reusability >Learnability
Willing to Use	Yes	Yes	Yes
Improvement in CM, RU and CD	Yes	Yes	Yes
Suitable Users	Beginners to experts, specifically beginners	Medium users to expert, beginners in simple applications	Beginners to Experts
Applicable Domains	Personalized services, data aggregation and analysis, but not high-precision, low-latency applications like smart obstacle avoidance	Wide-range and public IoT service	Fast prototyping, IoT Education
Transferable Knowledge	RESTful IoT Service Architecture	IoT Task flow design	SOA usage and IoT semantic description

third interviewee mentioned that the proposed framework "helps to improve the reusability, since it requires developers to separate the integrated IoT application into self-contained services and provide standard programmable interface for each service. If this requirement is achieved, the functionality of one IoT application will be definitely easier to be reused in other applications." Two out of three interviewees agreed that the proposed framework was suitable for developers from novice to expert level; While the remaining one thought it better suited medium IoT developers with basic programming skills. But he also conditionally agreed that in the case of simple application, proposed framework was friendly to beginners as well.

Possible application domains listed by interviewees included: personalized IoT service, wide-area data aggregation and analysis (e.g. vehicle data), public IoT service, IoT fast prototyping, IoT education etc. And the first interviewee also pointed out that the proposed framework might not be suitable for applications that requires "high-precision, low-latency control, such as intelligent obstacle avoidance". The interview results also indicated that transferable knowledge, including RESTful IoT service, service-oriented architecture usage, semantic IoT description and IoT development task flow design could be learned from proposed framework and further be applied in general IoT development domains.

The overall results are shown in 5.12, among which CM is the abbreviation for "customization", RU for "reusability", and CD for "cross-domain interoperability".

5.3 Architectural Assessment

Unlike that expertise requirement and kick-start barriers are compare items closely related to user experience, customization cost, reusability and cross-domain interoperability were more structural aspects. Since theres still few standard comparison metric within this relatively new area, we had to borrow some of the indexes from related fields, such as software engineering, distributed computing, service modeling and etc.

In the coming subsections, we continued to invite the experts to give ratings over the three aspects, revealing the structural difference between proposed framework and the other three competitors. In addition, a scalability test was also carried out.

5.3.1 Customization Cost

Software customization can be divided into three levels: derivation, configuration and personalization, which can be measured by the complexity of achieving core requirements in each level [106]. Microsoft summarized that software application customization can usually be achieved by three methods: (1) rewrite source code, (2) plug custom component into existing system, (3) use scripting (or other methods) to re-define business logic [107]. Customizability is considered especially important when facing: (1) Heterogeneity of market demands; (2) Customers demands of fast and varied response to their needs (3) Competition from other related vendors [108]. And service customization issue can be divided into five layers according to service oriented architecture: operational system/device, component, service, process, presentation [109].

Since in this research we only discuss the IoT composition system, and considering the fact that customization in this kind of system actually means composing/replacing service nodes (IoT devices and Web services) for specific business logic or task flow, we defined a customization cost metric on the basis of literature reviews, which contained 6 features. The calculation of thee final customization cost, CCS, was based on the following equations:

$CCS = w_{NC} \times F_n(S_{NC}) + w_{IC} \times F_n(S_{IC}) + w_{CRC} \times F_n(S_{CRC}) + w_{CMC} \times F_n(S_{CMC}) + w_{DRC} \times F_n(S_{DRC}) + w_{LRC} \times F_n(S_{LRC})$, while normalization function $F_n = (Value - MinValue)/(MaxValue - Value)$.

1. **Node Composability (NC)**.

   Program Constraint (0): To be able for composition, nodes (device/service) are required to use specific Language, SDK, OS.

   Template Constraint (1): To be able for composition, nodes are required to provide specific functions, parameters.

   Interface Constraint (2): To be able for composition, nodes are required to provide specific access interface.

2. **Interface Complexity (IC)**.

   Language Restriction (0): Service's access interface can only be used by specific program Languages.

   Model Restriction (1): Service's access interface can be used by any program language who is capable of correctly handle the specific data model, for example: JSON object.

Type Restriction (2): Service's access interface can be used by any program language who is capable of correctly handle the specific data type, for example: Integer.

3. **Configuration Rule Complexity (CRC)**.

   No Common Rules (0): Any node may have its own configuration rules.

   Uniform Rules (1): All nodes can be configured by selecting from a set of common rules and configuring parameters.

   Predefined Rules (2): All rules are predefined, only parameters need to be configured.

4. **Composition Method Complexity (CMC)**.

   Variable-level detail (0): To complete a composition, all the required variables, relations between variable and conditions must be set correctly. For example, to "turn on" a device, all variables used to initialize the device must be set one by one.

   Event-level detail (1): To complete a composition, a set of events and their trigger conditions are required to be set. Each event represents a set of variables, relations and conditions. For example, to "turn on" a device, an event named "turn on" should be triggered.

   Semantic-level detail (2): To complete a composition, a set of semantic representations are required to be set. A semantic representation represents a set of events with different names and/or parameters but has the same meaning, for example: "turn on" and "start" may be two events used by different devices, but are identical in semantics if they link to the same ontology.

5. **Device Replacement Complexity (DRC)**.

   SDK level revision required (0): When two devices use different SDKs, extra workload have to be paid to replace one for another.

   Service level revision required (1): When two devices use different services, extra workload have to be paid to replace one for another. A service may encapsulate several SDKs for various types of devices.

   Interface level revision required (2): When two devices use different Interfaces, extra workload have to be paid to replace one for another. The same Interface may be used by various types of services.

6. **Logic Revision Complexity (LRC)**.

   Re-Program (0): The only way to change the logic of a composition is to re-program, re-compile and re-deploy the application.

   Re-Deploy (1): To change the logic of a composition, the application must be halted, re-deployed and restarted.

   On-The-Fly (2): Composition logic can be changed by user input at run time and take effects immediately. Usually symbol, graph, or semantic based methods are used to record inputs.

Based on this scale, we listed the comparison on customization cost of proposed framework with other parallel researches as below:

Table 5.13: Comparison on Customization Cost

FEATURE (weight)	HomeAssistant (2013)	Node-Red (2013)	PubNub-EON (2015)	HSML (2016)
NC (0.1)	1	2	2	2
IC (0.1)	0	2	1	1
CRC (0.2)	2	0	1	1
CMC (0.2)	1	1	0	2
DRC (0.2)	1	2	0	1
LRC (0.2)	1	2	0	2
Customization Cost Scale	0.37	0.47	0.17	0.5

5.3.2 Reusability

Software reuse is using the previously developed software for building of newly developing system [110], which is a common strategy for organizations to improve productivity, insure quality, and save cost [111]. Since 1980's, different kinds of metrics have been proposed to measure the degree to which a software can be reused, including: reuse level metric [112], reuse metric for Object-Oriented systems [113], reuse library metric [114], FCM [115]. However, they either focused on non-distributed system, required too many detailed information (thus are not practical for real use), or just related to a certain aspect of reusability. In this

research, we carried out a literature review and selected major indicators to construct our own metrics for experts to evaluate reusability for distributed system, especially those based on service oriented architecture.

For reusability evaluation, we established a reusability metric comprised of 7 features and grouped by 3 categories: structure, interface and component. All features were selected based on a throughout literature review. Each feature used a 3-point or 5-point scale, with a larger point indicated a better performance. The calculation of the final reusability scale, RS, was based on the following equations:

$RS = w_C \times F_n(S_C) + w_D \times F_n(S_D) + w_{KR} \times F_n(S_{KR}) + w_{IH} \times F_n(S_{IH}) + w_{SS} \times F_n(S_{SS}) + w_{SD} \times F_n(S_{SD}) + w_SA \times F_n(S_{SA})$, while normalization function $F_n = (Value - MinValue)/(MaxValue - Value)$.

1. **Coupling (C)** [116].

   Monolithic (0): The system structure is always treated as a single unit and its individual parts cannot be manipulated.

   Dependent (1): Individual parts can be manipulated separately. however, the running of each part depends on other parts, thus hard to be replaced.

   Self-contained (2): Each individual part can be running separately and replaced easily [117].

2. **Reusability Dependency (D)**.

   OS (0): The reusage can only be achieved on specific operation system.

   Language (1): The reusage can be achieved on any operation system, but only support specific language [118].

   Protocol (2): The reusage can be achieved on any operation system, by any language, if only it satisfies certain protocols.

3. **Knowledge required for reuse (KR)**.

   Source Code Reusability (0): Reuse can only be achieved by modifying the source code of the target system [119].

   White Box Reusability (1): Reuse can be achieved by using exposed interface of target system, without modifying the source code. However, one must be capable of reading source code to understand how to use the interfaces [120].

   Black Box Reusability (2): Reuse can be achieved without any knowledge of the source code, the only knowledge needed is the interface description of the target system [121].

4. **Interface Heterogeneity (IH)** [122].

   Heterogeneous (0): Each component may define its own interface specifications. The total number of the interfaces required for reuse is unlimited.

   Standardized (1): All components can be reused by using limited types of interfaces.

   Uniform (2): All components can be reused by using the same type of interface.

5. **Service Source (SS)** [123].

   Internal (0): Reusable components only come from the internal system.

   Global (1): Reusable components come from both internal and external systems.

   Participatory (2): Reusable components come from both internal and external systems, and a composition of reusable components from internal and external systems is also a reusable component.

6. **Service Discoverability (SD)** [123].

   Standalone (0): Components are provided separately without relationship.

   Networked (1): Components are linked with each other, description files contain linkage information are provided by each component.

   Indexed (2): Components are indexed in one or more central portals.

7. **Service Adaptability (SA)** [121, 123].

   Non-adaptable (0): Components can only be used for predefined contexts.

   Adaptable (1): Components can be adapted to varied use contexts by manual setting.

   Adaptive (2): Components can be adapted to varied use contexts by giving description files.

   Semantic (3): Components can be adapted to varied use contexts by giving semantic description files.

   Automatic (4): Components can be adapted to varied use contexts automatically.

Based on this scale, we listed the comparison on reusability of proposed framework with other parallel researches as below:

Table 5.14: Comparison on Reusability

FEATURE (weight)	HomeAssistant (2013)	Node-Red (2013)	PubNub-EON (2015)	HSML (2016)
C (0.2)	1	1	1	1
D (0.1)	1	1	2	2
KR (0.1)	1	2	0	2
IH (0.2)	2	1	1	2
SS (0.1)	0	2	1	2
SD (0.1)	2	2	2	1
SA (0.1)	1	1	0	1
Reusability Scale	0.37	0.41	0.3	0.47

5.3.3 Cross-Domain Interoperability

IEEE defines interoperability as, the ability of two or more systems or components to exchange and use the exchanged information in a heterogeneous network [124].The US Department of Defense defines interoperability as, the ability of systems, units, or forces to provide services to and accept services from other systems, units, or forces, and to use the services so exchanged to enable them to operate effectively together [125].

$CDIS = w_{DID} \times F_n(S_{DID}) + w_{DI} \times F_n(S_{DI}) + w_{OA} \times F_n(S_{OA}) + w_{SD} \times F_n(S_{SD}) + w_{SIP} \times F_n(S_{SIP}) + w_{SS} \times F_n(S_{SS})$, while normalization function $F_n = (Value - MinValue)/(MaxValue - Value)$.

1. **Device Interoperation Dependency (DID)** [126–128].

   Operating system (0): To operate device across domain, specific operating system must be used.

   Network protocol (1): To operate device across domain, specific network protocol must be used.

   Virtualization Technology (2): To operate device across domain, specific virtualization technology (for example: java vm) must be used.

   Syntactic standard (3): To operate device across domain, specific syntax and encoding standard (for example: bit table) must be used.

   Semantic standard (4): To operate device across domain, agreed-upon semantic standard must be referred to.

2. **Data Interoperability (DI)**.

   Domain-specific (0): To understand data from another domain, (human or machine) user must know the domain-specific method to process raw data.

   Formatted (1): Data from another domain are formatted according to open formats, for example: xml, json, csv, etc.

   Standardized (2): Data from another domain can be interpreted and understood by using open standards, for example: SWIFT in financial industry, sensorML in sensing industry.

3. **Object Abstraction (OA)** [129].

   Technology (information) level abstraction (0): The physical device can be mapped to and manipulated by a corresponding abstract entity (i.e. service, in this research) in another domain. However, the abstract entity has to expose detailed technology information of the physical object, such as: internal data model, hardware ports, sockets, etc.

   Functional level abstraction (1): The abstract entity has to expose function information, such as: function name, parameters, etc. The technology level details are unnecessary.

   (programmatic) Logic level abstraction (2): The abstract entity only need to expose logic level information, such as: status, events, conditions, etc. The technology and function level details are unnecessary.

4. **Service Description (SD)**.

   None (0): No service description information.

   Comment (1): Description information is provided as comments in source code files or other text files.

   Separate (2): Description information is provided in standardized file (for example: wsdl) as separate elements, such as: address, usage, properties, functionalities, status, events, etc.

   Connected (3): Description information is provided in standardized file as elements and relations between elements.

   Modeled (4): Description information is provided in standardized file as elements and relations by using a common model, for example: UML, State Transition Model, Event-driven Process Chain, etc.

5. **Service Interoperation Prerequisite (SIP)**.

   Source code (0): To interoperate services between domains, each others source code must be retrieved and understood.

   Human readable description (1): To interoperate services between domains, the only prerequisite is to retrieve and understand each others service description file. The service description file is only human readable.

   Human-Machine readable description (2): To interoperate services between domains, the only prerequisite is to retrieve and understand each others service description file. The service description file is both human and machine readable.

6. **Service Statelessness (SS)**.(means the service treats each request as an independent transaction that is unrelated to any previous request, whether by the same service consumer or any other service consumer, so it does not need to wait others to finish related steps)

   Non-statelessness (0): None of the services are stateless.

   Entity service statelessness (1): Entity services are stateless. Entity services are those services interacting with resources, for example: IoT devices.

   Task service statelessness (2): Both entity and task services are stateless. Task services are those services storing task logic, for example: composition service.

Based on this scale, we listed the comparison on customization cost of proposed framework with other parallel researches as below:

5.3.4 Scalability

To simulate the real cloud deployment and usage scenario, we established a testbed based on a cloud platform at jcloud.sjtu.edu.cn as shown in 5.6. The cloud platform itself was developed by OpenStack, an open source cloud computing software. We created two virtual networks on OpenStack, one to simulate servers network (the ServerNet), one to simulate clients network (the ClientNet). In cloud computing environment, the virtual server is also a web service which can be dynamically created and merged into service pool for elastic service expansion. In this situation, the service scalability is theoretically unlimited. But in real practice, the hardware capacity and communication cost, i.e. the real computing hardware

Table 5.15: Comparison on Cross-Domain Interoperablity

FEATURE (weight)	HomeAssistant (2013)	Node-Red (2013)	PubNub-EON (2015)	HSML (2016)
DID (0.3)	2	2	3	4
DI (0.3)	1	1	2	2
OA (0.1)	2	2	1	2
SD (0.1)	3	1	1	3
SIP (0.1)	2	1	0	1
SS (0.1)	0	0	1	1
Interoperablity Scale	0.41	0.34	0.5	0.63

capacity and communication latency will inevitably reach their limitation. We'll discuss this under our horizontal and vertical scalability architecture in real cloud platform deployment.

In our cloud platform, we used service replication as a basis to provide horizontal scalability. When one service became the bottleneck of the whole system, we could replicate this service and use load balance method to redirect requests to replicated services accordingly. This method could expand the service capacity horizontally, as we will evaluate in the following test. The horizontal scalability architecture has its advantages, such as: software defined, highly flexible and elastic. But it also has its limitations, which mainly caused by extra transaction costs between replicated services, and the fact that the hardware especially the network bandwidth cannot be expanded without limitation in a single geographical location. If the horizontal scalability architecture reached its limitation, we recommend to use vertical scalability architecture for further expansion, which means to deploy more cloud platforms that are physically and geographically separated.

In this section, we focused on evaluating horizontal scalability in our system because the vertical scalability was usually addressed by the infrastructure design and hence was out of our research scope. More specifically, in our virtual serverNet, we could deploy service servers dynamically by service replication, for example: orchestration service and message broker. For services of the same type, a service pool will be created for management. A load balancing service can be in charge of managing the communication traffic in service pool and redirecting requests from clientNet to idle services to avoid bottleneck caused by a specific service.

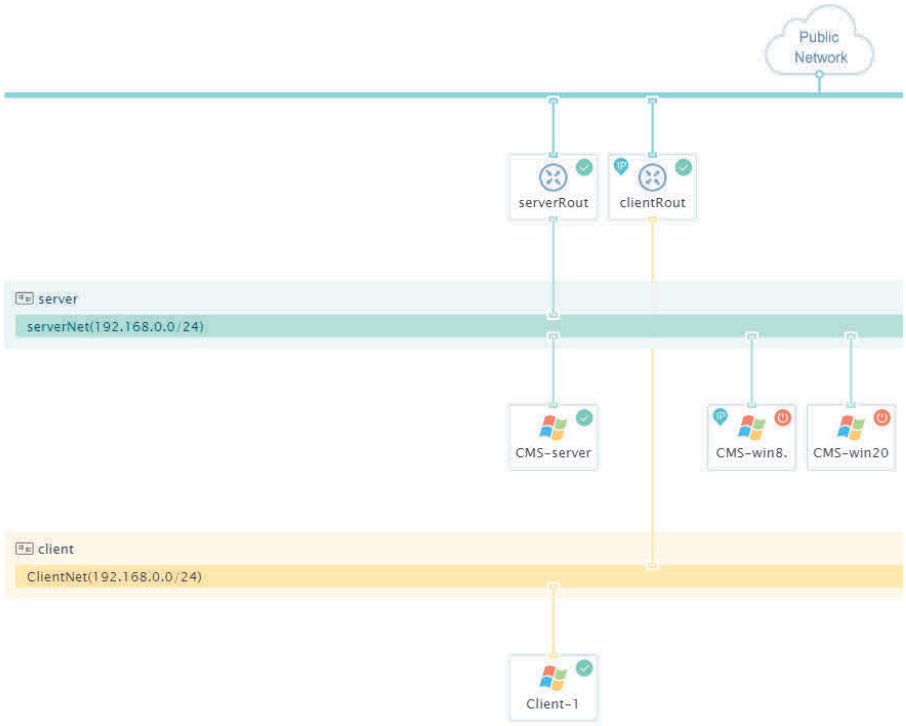


Figure 5.6: Testbed on Cloud Server Environment

In the following test, we managed to evaluate the performance of using service replication method for horizontal scalability. The service resource pool used in this test contained two message broker services. The reason why we choose message broker service to test system performance was because, most communication traffic in our system was related to message broker who was very likely to become a bottleneck as requests increased. However, this did not mean only message broker service can be replicated for system scalability. Any service that may become the bottleneck could be scaled up by using the same technique we introduced above. The Round-Robin algorithm was used to control load balance for the service pool.

In client side, a test tool developed by node.js was used to simulate simultaneous requests from 5-100 device nodes, and each device was supposed to send 10 requests per second. The requests were delivered via actual internet to reach server side to fully simulate the real cloud scenario. The test was carried out twice for comparison: In test 1, only one message broker was adopted to manage the service pool; While in test 2, two replicated message brokers were adopted.

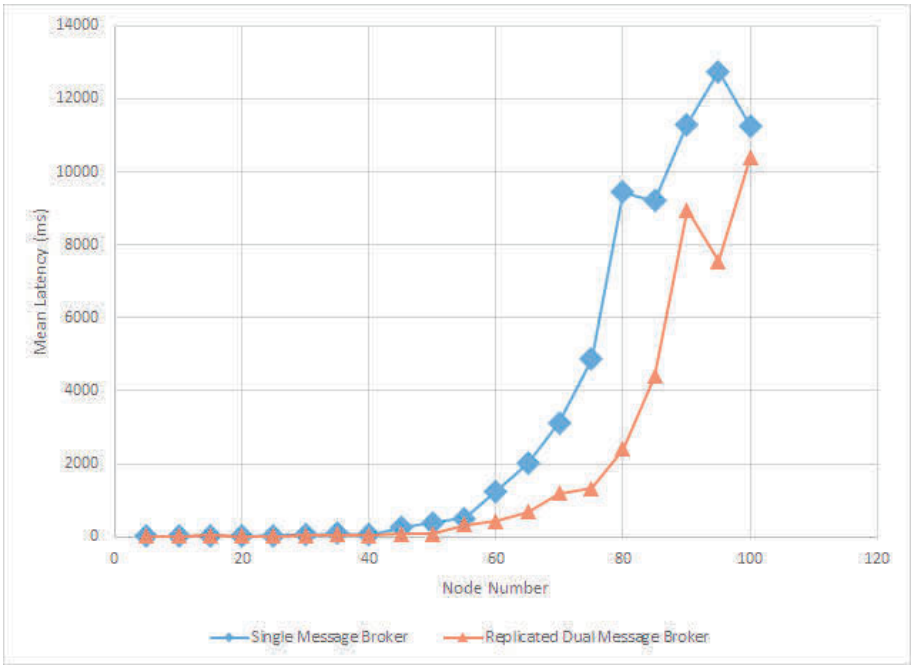


Figure 5.7: Test Result: Mean Latency

In both tests, the mean latency and the rejection rate increased steeply when the amount of simultaneous device nodes crossed a threshold which indicated the system capacity was reached. The difference was, by replicating and providing just one more message broker in the service pool, the threshold could be scaled up from 65 to 80 (judging from mean latency solely) or from 60 to 85 (judging from error rate solely). Hence, we could draw a conclusion that, the service replication method we proposed was applicable for providing horizontal scalability. Each service replication was supposed to support extra 20 devices more. Considering the fact that, in real use scenario, most IoT Web services will not send state messages (not raw data) 10 times per second, the actual gain will be more than 20 devices. Predictably, an increment of 200 devices is affordable if the message delivered from each service is set at a rate of 1 per second (which is very common for a IoT service).

Hence, in cloud platform, we can easily deploy horizontal scalability architecture to support thousands of devices in a single composition task. If a composition task contains more devices that are geographically dispersed, vertical scalability method can also be introduced to further deal with large scale requests by pro-

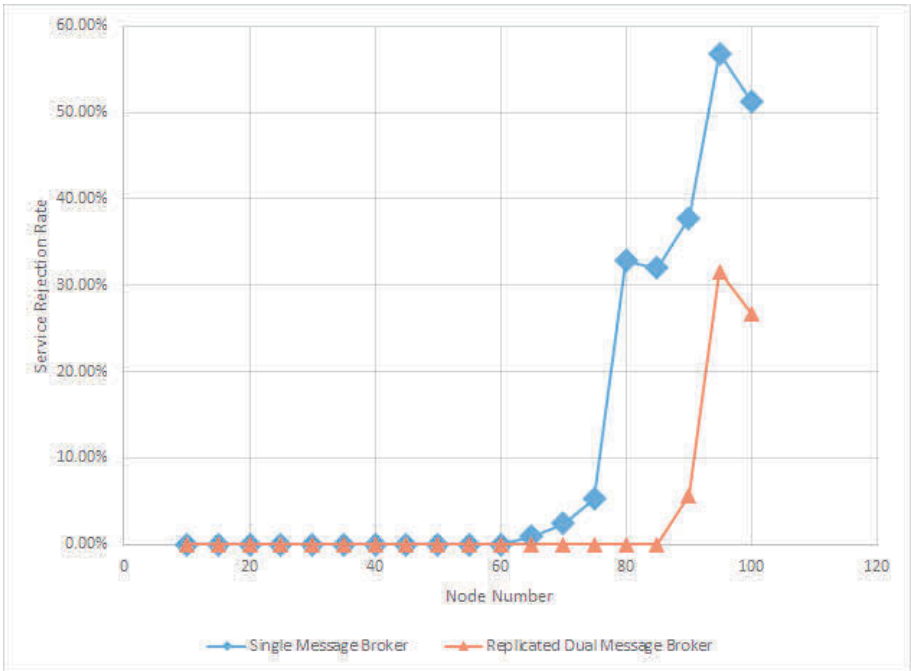


Figure 5.8: Test Result: Rejection Rate

viding physically and geographically distributed cloud servers.

Another solution is to deploy the message broker at run time on client side. In this case, communication between client and composition server only takes place when the client submits a composition task to central orchestration service for the first time. And the central orchestration service will in return reply with generated composition logic and configure the message broker located at the client side. The communication cost in this phase approximates to a constant and can be omitted when compared with the request/response between message broker and resources to be composed. As it greatly depends on the network environment of client side, we have simulated message broker deployed on a variety of hardware and systems, to send requests to a target resource, which may either located on a cloud server or an edge device.

We estimated the average time consumed from message broker initiates a request to the response of target resource bounces back. The test were repeated every 5 seconds for 10 times. The cloud server specification was already described in the beginning of this section, while the edge device was deployed on a Raspberry Pi and shared the same network segment as the clients who used WLAN

and Ethernet connections. Average round-trip duration was shown in Table 5.16.

Table 5.16: Roundtrip Duration (ms) in Cloud and Edge Computing Environment

system	Windows (10, 64bit)	
hardware	I7-5500U(2.40GHz*2), 16G RAM	
network	1.0Gbps Ethernet	
	Cloud	Edge
Chrome	131.6	75.1
Fireforx	133	107.1
Edge	134.1	68.9
system	MacOS (10.11)	
hardware	I5 2GHz, 8G RAM	
network	144.0 Mbps WLAN	
	Cloud	Edge
Safari	272	161.8
system	Android (7.0)	
hardware	MSM8994(2G*4+1.5G *4), 3G RAM	
network	LTE	
	Cloud	Edge
Chrome	333.1	333.4
Firefox	275.7	339.4

This case only imitated the simplest scenario of one message broker managing one service. The overall cost will however rise as the number of involved resource nodes increases. The communication pattern (synchronous/asynchronous) is also another relevant factor and will turn the duration estimation in multi-nodes composition scenario into a more complicated issue.

5.4 Discussions and Limitations

The user experiments, expert rating and architectural comparison shown the overall performance of proposed framework outscored its mainstream counterparts, especially in Reusability and Cross-Domain Interoperability. In Customization Cost and Expertise Requirement, it at least equals the best competitor, if not better. Its also proven to be very easy to learn and enjoyable to use, and can be scaled easily by simple service replication strategy. Below we will discuss the as-

sessments of each research issue in more detail, and also summarize the limitation identified in the evaluation processes.

As evaluated in user experiment and expert interview, the proposed framework has obvious advantages over two of the three mainstream frameworks in Expertise Requirement and Kick-start Barrier assessment, statistical significance was found in most rating items, including: time consumption, Problems Asked, NASA-TLX Overall Rating, Mental Demands and Effort. As comparing with the last one, Node-Red, the score is very close. While we believe a major reason (that the advantages of proposed framework dont shown obviously) is because the user experiment is highly time consuming and work intensive, to prevent participants from feeling exhausted, we only selected the simplest tasks. In a more complicated task, the uniform description and usage method of all resources that supported by proposed framework will generate more obvious advantages over frameworks using heterogeneous description and usage methods, such as Node-Red. As a proof, in free question stage, two participates selected the proposed framework as the most willing to use framework in the future. The reason given by participates is that there is no need to learn how to use different components because they all follow the same usage model. Also in expert interview, two experts agreed that the proposed framework do help to lower down the expertise requirements and kick-start barrier because of its neutrality (e.g. platform, programming language, standard), flexibility (e.g. changing nodes or task logic during run-time) and highly transferable knowledge (e.g. FSM model, SOA). Besides the selection of experiment tasks (which only contains simple tasks), the number of participates (only 4) is another limitation in user experiment. Ideally, a test on tens or hundreds of users will bring more convincing conclusion. However, its beyond our capability right now.

For Reusability, Interoperability and Customization Cost assessment, both architectural comparison and expert interview shown the proposed framework can provide better support due to some unique features, including: loose-coupling, no technical knowledge required, uniform component interface, etc. As comparing with mainstream competitors, the proposed framework has a better overall rating. Some experts mentioned that when they decide to use a framework or not, those features are very important because the application itself is not the purpose, test the development process iteratively to prove ones idea is, while the proposed framework can satisfy them from several key aspects, including: easy-to-replace components based on FSM model, on-the-fly composition mechanism, etc.

It is worthy to mention that, to avoid bringing comparison too many trivial, we only used 3 or 5 degrees in rating, which make the comparison a little rough. Thus some detailed features may not be fully represented in the comparison. For example, for coupling comparison, the actual situation may be more complicated than our assessment. Even if two frameworks both have individual parts that can be manipulated separately, the statelessness of each part can further lower down the coupling. These minor details are not reflected in architectural comparison due to complexities. However, the proposed framework has advantages on all these detailed features (so that it will not bring bias into assessments). The detailed technical analysis can be found in previous chapters.

Another limitation of architectural comparison may come from the representativeness of the items used in the assessment. Although each item is carefully selected from literature review on previous research or related fields, there is still possibilities that some items are lacked or some of them are not so proper for IoT field. Since theres no commonly accepted rating standard at this moment, we have to leave the judgment to the readers.

5.5 Summary

In this chapter, a comprehensive evaluation for proposed open IoT service composition framework was carried out to measure to what extent improvements had been achieved on expertise requirement, customization cost, reusability and cross-domain interoperability. The overall evaluation strategy contained three parts: 1) User test, 2) Expert interview and 3) Architectural comparison.

In Section 5.1, a series of beginner-centered user tests were conducted specifically for estimating expertise requirement and kick-start barriers when using proposed web development toolkit, HSML. We tested over learnability, socialbility, retrievability and task load. Results showed that compared with mainstream open IoT service composition frameworks, HSML was among the best development toolkit in regard to overall task load performance. We also found that the HTML-like domain-specific language syntax also helped provide beginner-affordable learnability, increase the willingness of sharing composition results, as well as improve the retrievability of IoT resources by rich user-generated semantic tags.

In Section 5.2, we conducted a complementary expert interview to collect feed backs from veteran developers inside the industry. Three experienced experts

from different IoT tracks were invited to answer questions about: 1) Interviewees' personal technical background, 2) Interviewees' preferences on general features of IoT development frameworks, and 3) Feed backs after trying out the proposed framework. Results showed that all three interviewees gave positive answers, when asked if they were willing to use proposed framework in the future. The interviewees also reached a consensus that proposed framework had improved customization cost, reusability and cross-domain interoperability, compared with other frameworks and platforms they previously used.

In Section 5.3, in order to reveal the structural difference between proposed framework and the other competitors, we continued to invite the experts to give ratings over 19 indexes traversing customization cost, reusability and cross-domain interoperability aspects. For each index, a 3-point or 5-point rating scale was adopted and total weighted arithmetic means were calculated. Results showed that proposed framework had the best performance on cross-domain interoperability while shared similarly leading results with another framework on customization and reusability. In addition, we also accessed the scalability of proposed framework on virtual server/client networks, in which mean latency, rejection rate and round trip duration were tested. And result shown that proposed framework was able to support 60 to 80 nodes which send 10 requests per second, by each single message broker. And by replicating message broker, the scalability can be improved at the rate about 20 nodes with each extra broker.

Lastly, in Section 5.4, we had a general discussions about in what range the overall evaluation results were considered valid. Some limitations found in evaluation approach and experiment design were also listed.

Chapter 6
Conclusion

6.1 Contribution

It is said that by 2020, there will be 50 to 100 billion things connected to the Internet [130]. IoT support penetrates almost every aspect of society, e.g. municipal governance, economy, mobility, environment and living, fostering a wide spectrum of applications ranging from transportation, public welfare, sustainability, tourism, business all the way to city safety [131]. These actual needs drive IoT services to overcome the technical and organizational boundaries, particularly in areas like smart city, environmental intelligence and etc.

On the other hand, The World Wide Web built upon the Internet is hitherto the most effective open platform for everyone to share human perceived reality or virtual reality globally. And the introducing of sensors and actuators is supposed to add real-world data, and optionally awareness to the Internet. However, this deceptively simple addition is a transformational change, given that current web infrastructure itself was not prepared with the motility to grasp the physical environment information spontaneously [132].

The Web has been providing user interface of simplicity and generality. It is also expected to be consistent with sensors and actuators all along, especially when the popularity of built-in sensors/actuators and smart devices has made everyone both the provider and the consumer of physical information at the same time. Kick-start barriers and learning cost will greatly affect how voluntary the users, i.e. the readers, authors and application developers, participate in the creation and structuring of information. And if we refer sensors/actuators to a new kind of hypermedia contents, we are actually expecting that sensors/actuators and their functionality can be retrieved, discovered, integrated and reused by various applications over the Internet, in the same manner as we has been utilizing plain texts, images, videos, audios and so on nowadays.

As a result, the mutual needs from both IoT service enablement and Web

infrastructure evolvement have greatly reshaped the domain requirements of sensor/actuator/IoT web applications development:

1. **Internet-scale**. Nowadays, the Internet works as the information infrastructure that provides interconnectivity of distributed ICT devices and information networks that are not only dispersed geographically, but also across multiple organizational boundaries. Examples like sensor cloud platform (AWS IoT, SAP HANA), open sensor portal (OGC SWE, sensorPedia), and IoT application.

2. **Ubiquitous accessibility**. Suppliers of information services are supposed to provide ubiquitous and constant accessibility of data and resources that are time and/or space sensitive. Particularly, for environment monitor sensors, it usually does not refer to the accessibility of one specific sensor node, but rather the availability of sensor data from peer sensor groups within a certain range of time and space.

3. **General purposes**. Despite traditional machine-to-machine control, geographic observation, military and defense purposes etc, sensors and actuators are gaining a booming portion of consuming electronics and end-user-centered market. Responsively, non-dedicated applications have appeared with more general purposes that do not exclusively rely on dedicated platforms, operating systems or devices.

4. **Context adaptable**. Modern sensor-actuator applications are supposed be deployed in volatile physical environments along with varied computing and user contexts, which consequently requires the ability of changing business logic in-situ to adapt to context transition, either by predefined active methods or runtime passive methods.

The newly rising domain requirements emphasize on Web based ubiquitous system features more than ever, e.g. interoperability, agility, reusability and participatory, etc. As our target, this specific research is dedicated to provide a scalable, platform-independent, general-purpose open framework with fine-grained, light-weighted general development interface for Internet-scale, cross-organizational IoT services. Oriented to most mainstream sensor/actuator/IoT web resources (specifically RESTful and COAP-based resources), we have further extended state-based resource interface into a state machine model and provided standard descriptions

based on the combination of SensorML and SCXML. It leads to a richer but lucid expression of exposed functionality and control mechanisms, which used to be confined to only well-trained technicians and domain experts. It has increased the composability of each component resource and the overall degree of automatability. Consistent paradigm based on state transfers has also been adopted by the central service orchestration to integrate diverse IoT web resources together with large amounts of existing web services, which contributes to the reusability of legacy system and current technology stack. Careful estimation of scalability, applicability and general performances in a variety of computing environments has been conducted. And proposed framework and its corresponding Web API have been proved to be capable of loosen the tight coupling among service components and lower down the overall development cost in a wide range of general application scenarios. It is supposed to be adaptable to multiple distributed computing environments and capable to generate certain level of smartness to complete predefined task or business logic in an automatic or semi-auto manner, the development of which also will rely on the maturing semantic web and machine learning technology in future.

As we already suggested in previous research [133], the proposed framework allows the reusability of existing technology stack and legacy computing systems, significantly lowers down the technical threshold for sensors/actuators entering current web computing systems, while not letting any of current deployed web services to be degraded. This contributes to the future Internet as a ubiquitous network of interconnected objects that not only harvests information from the environments (sensing) and interacts with the physical world (actuation/command/control), but also uses existing Internet standards to provide services for information transfer, analytic, and applications [130]. And hopefully, it will pave the road towards "the Equity of IoT service", that each and every citizen shall have equitable, inclusive accessibility and quality of public IoT infrastructure. Going hand in hand with complementary technologies, e.g. semantic web, machine learning and blockchain, etc., this research will become the primary step towards an open, trustable, and autonomous smart society.

6.2 Limitation

The major limitations of this research lied in both the architectural and methodological aspects. The former was the common restrictions that shared by most

frameworks that based on IoT service composition approach, while the latter was the limitations we found in regard to the concrete methodology adopted in this research.

1. **Architectural Limitation**. As already mentioned in Chapter 1, virtualization and servitization are among the prerequisites for IoT service composition to convert vendor-specific, device-dependent functionality into unified, composable development interfaces with open accessibility. Introducing this kind of abstract, intermediate layers will inevitably entails extra computational cost and real-time latency. Despite limited IoT devices, the servitization of most resource-constraint sensor systems, e.g. wireless sensor networks, are taken over and accomplished by either some sink node or IoT cloud platform, instead of the device itself, which consequently generates a longer response time. Therefore, the proposed framework may not be applicable to build latency-sensitive applications, such as smart obstacle avoidance, multi-sensory detection. Fortunately, servitization on resource-constraint devices become feasible thanks to the maturity of counterpart web technology stack like EXI, CoAP and 6LoWPAN etc. Together with the rapid development of micro-controllers and high-speed wireless Internet accessibility, it is supposed to reduce the overall communication cost to some extent.

2. **Methodological Limitation**. In this research, we have adopted a state-transfer-based IoT service composition approach and proposed Finite-State-Machine-based IoT service modelling. An IoT service composition is actually equivalent to a serial combination of two or multiple FSM-modelled IoT nodes. And theoretically, the composition as a whole is composable and can further be nested into hierarchical compositions [134]. However, it is very difficult to estimate and handle the computational complexity and possible state message blockage of hierarchical composition at current stage. And a more comprehensive mathematical model must be established, so that the proposed FSM-based service modelling can be applied to formally verify the correctness inside a workflow specification, and further support automatic composition of Finite-State Machines.

6.3 Future Issues

In the coming future, we will further coordinate with some partner technologies in relation to IoT service development, and extend the proposed framework in following aspects

1. **Process Automation**. Given that we have already proposed a machine-readable resource description (StateML) for IoT web services, in next step, we consider to provide a corresponding mechanism in HSML toolkit that can read StateML files and automatically extract necessary information. So that actions like specifying resource accesses, or simple measure conversion, can be further taken over by HSML, which is supposed to reduce manual intervention in business logic building and enhance the overall degree of process automation.

2. **Cross-domain Discoverability**. In 5.2.3, we have showed that the rich semantic relations contained in user-generated HSML texts helped improve the retrievability of IoT resources. The proposed framework is considered to be innately consistent with technologies like linked services and semantic web. It is plausible to adopt, for example, a SPARQL-based query mechanism for IoT developers to discover and query over IoT services at a semi-semantical or semantical manner. Moreover, Graph database is another adoptable partner technology, which enables graph-structured storage of interconnected information, to push cross-domain discoverability of proposed framework one step further.

3. **Security**. The proposed framework intends to encourage developers to reuse existing IoT services and share their own services. Therefore it becomes an urgent need to address issues like how to identify if a third-party service is reliable or not, how to ensure each and every operation within a composition task is validated and traceable, and how to regulate service accessibility according to different level of user authority and etc. Some inspirations may be found in related researches on IoT blockchain and access delegation etc.

Other future functional improvements may exist in offering a hybrid user interaction by combining graphic element and domain-specific language together. However, more consideration must be taken in regard to the balance between learning cost and expressiveness.

6.4 Summary

Going hand in hand with partner technologies like semantic web, maching learning and blockchain etc., this research will hopefully be the primary step to achieve the equity of IoT service as our long-term goal. IoT will eventually become part of future public infrastructure as well as part of our future society. By then we believe that each and every citizen shall have equitable, inclusive accessibility and service quality provided by the future IoT. And our research will ultimately contribute to an open, trustable and autonomous smart society.

As a conclusion, we discussed the contributions, the limitations which included both architectural and methodological aspects, as well as the future issues in this chapter. This research was dedicated to: 1) implement the proposed state-transfer-based open IoT service composition framework, 2) demonstrate the advancement by developing a few target domain applications, e.g. multi-source environmental monitoring, open automation systems, and etc., 3) evaluate the improvement in expertise requirement, customization cost, reusability and cross-domain interoperability. And the results indicated a better overall performance than other mainstream IoT service composition frameworks.

References

[1] Salem Hadim and Nader Mohamed. Middleware: Middleware challenges and approaches for wireless sensor networks. *IEEE distributed systems online*, 7(3):1–1, 2006.

[2] Kay Römer, Oliver Kasten, and Friedemann Mattern. Middleware challenges for wireless sensor networks. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):59–61, 2002.

[3] Abdelmounaam Rezgui and Mohamed Eltoweissy. Service-oriented sensor–actuator networks: Promises, challenges, and the road ahead. *Computer Communications*, 30(13):2627–2648, 2007.

[4] Arne Bröring, Johannes Echterhoff, Simon Jirka, Ingo Simonis, Thomas Everding, Christoph Stasch, Steve Liang, and Rob Lemmens. New generation sensor web enablement. *Sensors*, 11(3):2652–2699, 2011.

[5] Stefan Ferber. How the internet of things changes everything. *Harvard Business Review*, 2013.

[6] Scott Loveland, Eli M Dow, Frank LeFevre, Duane Beyer, and Phil F Chan. Leveraging virtualization to optimize high-availability system configurations. *IBM Systems Journal*, 47(4):591–604, 2008.

[7] Flávio Ramalho and Augusto Neto. Virtualization at the network edge: A performance comparison. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2016 IEEE 17th International Symposium on A*, pages 1–6. IEEE, 2016.

[8] Takayuki Suyama, Yasue Kishino, and Futoshi Naya. Abstracting iot devices using virtual machine for wireless sensor nodes. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pages 367–368. IEEE, 2014.

[9] Sarfraz Alam, Mohammad MR Chowdhury, and Josef Noll. Senaas: An event-driven sensor virtualization approach for internet of things cloud. In *Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on*, pages 1–6. IEEE, 2010.

[10] Jan S Rellermeyer, Michael Duller, Ken Gilmer, Damianos Maragkos, Dimitrios Papageorgiou, and Gustavo Alonso. The software fabric for the internet of things. In *The Internet of Things*, pages 87–104. Springer, 2008.

[11] Michael P Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: a research roadmap. *International Journal of Cooperative Information Systems*, 17(02):223–255, 2008.

[12] Richard Hall, Karl Pauls, Stuart McCulloch, and David Savage. *OSGi in action: Creating modular applications in Java*. Manning Publications Co., 2011.

[13] Matthias Thoma, Sonja Meyer, Klaus Sperner, Stefan Meissner, and Torsten Braun. On iot-services: Survey, classification and enterprise integration. In *Green Computing and Communications (GreenCom), 2012 IEEE International Conference on*, pages 257–260. IEEE, 2012.

[14] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. Web services. In *Web Services*, pages 123–149. Springer, 2004.

[15] Quan Z Sheng, Xiaoqiang Qiao, Athanasios V Vasilakos, Claudia Szabo, Scott Bourne, and Xiaofei Xu. Web services composition: A decades overview. *Information Sciences*, 280:218–238, 2014.

[16] Gabriel Montenegro, Nandakishore Kushalnagar, Jonathan Hui, and David Culler. Transmission of ipv6 packets over ieee 802.15. 4 networks. Technical report, 2007.

[17] Jonathan Hui and Pascal Thubert. Rfc 6282 compression format for ipv6 datagrams over ieee 802.15. 4-based networks. sep-2011, 2011.

[18] A Castellani, Salvatore Loreto, Akbar Rahman, Thomas Fossati, and Esko Dijk. Best practices for http-coap mapping implementation. *IETF work in progress*, 2012.

[19] Zach Shelby, Klaus Hartke, and Carsten Bormann. The constrained application protocol (coap). 2014.

[20] Andrew Banks and Rahul Gupta. Mqtt version 3.1. 1. *OASIS standard*, 2014.

[21] John Schneider, Takuki Kamiya, Daniel Peintner, and Rumen Kyusakov. Efficient xml interchange (exi) format 1.0. *W3C Proposed Recommendation*, 20, 2011.

[22] Angelo P Castellani, Nicola Bui, Paolo Casari, Michele Rossi, Zach Shelby, and Michele Zorzi. Architecture and protocols for the internet of things: A case study. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, pages 678–683. IEEE, 2010.

[23] Mike Botts, George Percivall, Carl Reed, and John Davidson. Ogc® sensor web enablement: Overview and high level architecture. In *International conference on GeoSensor Networks*, pages 175–190. Springer, 2006.

[24] Irma Morrison Alan Freedman. Application framework, 2017.

[25] wikipedia. Software framework, 2017.

[26] Rodger Lea, Simon Gibbs, Alec Dara-Abrams, and Edward Eytchison. Networking home entertainment devices with havi. *Computer*, 33(9):35–43, 2000.

[27] Rudolf HJ Bloks. The ieee-1394 high speed serial bus. *Philips Journal of Research*, 50(1):209–216, 1996.

[28] UG OPENHAB. Openhab.

[29] W3C. W3c mission.

[30] Tim Berners-Lee. A short history of "resource" in web architecture, 2009.

[31] Maurice Ter Beek, Antonio Bucchiarone, and Stefania Gnesi. Web service composition approaches: From industrial standards to formal methods. In *Internet and Web Applications and Services, 2007. ICIW'07. Second International Conference on*, pages 15–15. IEEE, 2007.

[32] Deze Zeng, Song Guo, and Zixue Cheng. The web of things: A survey. *JCM*, 6(6):424–438, 2011.

[33] Nils Glombitza, Dennis Pfisterer, and Stefan Fischer. Ltp: An efficient web service transport protocol for resource constrained devices. In *Sensor Mesh and Ad Hoc Communications and Networks (SECON), 2010 7th Annual IEEE Communications Society Conference on*, pages 1–9. IEEE, 2010.

[34] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1, 2000.

[35] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web services description language (wsdl) 1.1, 2001.

[36] Tom Bellwood, Luc Clément, David Ehnebuske, Andrew Hately, Maryann Hondo, Yin Leng Husband, Karsten Januszewski, Sam Lee, Barbara McKee, Joel Munter, et al. Uddi version 3.0. *Published specification, Oasis*, 5:16–18, 2002.

[37] Diane Jordan, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, et al. Web services business process execution language version 2.0. *OASIS standard*, 11(120):5, 2007.

[38] Scott de Deugd, Randy Carroll, Kevin Kelly, Bill Millett, and Jeffrey Ricker. Soda: Service oriented device architecture. *IEEE Pervasive Computing*, 5(3):94–96, 2006.

[39] Antonio Pintus, Davide Carboni, Andrea Piras, and Alessandro Giordano. Connecting smart things through web services orchestrations. *Current Trends in Web Engineering*, pages 431–441, 2010.

[40] François Jammes and Harm Smit. Service-oriented paradigms in industrial automation. *IEEE Transactions on Industrial Informatics*, 1(1):62–70, 2005.

[41] Luciana de Souza, Patrik Spiess, Dominique Guinard, Moritz Köhler, Stamatis Karnouskos, and Domnic Savio. Socrades: A web service based shop floor integration infrastructure. *The internet of things*, pages 50–67, 2008.

[42] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation, 2000.

[43] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. big'web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814. ACM, 2008.

[44] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of things for smart cities. *IEEE Internet of Things journal*, 1(1):22–32, 2014.

[45] Thomas Luckenbach, Peter Gober, Stefan Arbanowski, Andreas Kotsopoulos, and Kyle Kim. Tinyrest-a protocol for integrating sensor networks into the internet. In *Proc. of REALWSN*, pages 101–105, 2005.

[46] Sami Mäkeläinen and Timo Alakoski. Fixed-mobile hybrid mashups: Applying the rest principles to mobile-specific resources. In *International Conference on Web Information Systems Engineering*, pages 172–182. Springer, 2008.

[47] Adam Dunkels et al. Efficient application integration in ip-based sensor networks. In *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 43–48. ACM, 2009.

[48] V Dambal. Rest-ful services, 2010.

[49] Mikel D Petty and Eric W Weisel. A composability lexicon. In *Proceedings of the Spring 2003 Simulation Interoperability Workshop*, volume 2003, pages 181–187, 2003.

[50] Paul K Davis and Robert H Anderson. Improving the composability of dod models and simulations. *The Journal of Defense Modeling and Simulation*, 1(1):5–17, 2004.

[51] Andreas Tolk. Interoperability and composability. *Modeling and simulation fundamentals: theoretical underpinnings and practical domains*, pages 403–433, 2010.

[52] Shankar R Ponnekanti and Armando Fox. Sword: A developer toolkit for web service composition. In *Proc. of the Eleventh International World Wide Web Conference, Honolulu, HI*, volume 45, 2002.

[53] Torsten Dinsing, G Eriksson, Ioannis Fikouras, Kristoffer Gronowski, Roman Levenshteyn, Per Pettersson, and Patrik Wiss. Service composition in ims using java ee sip servlet containers. *Ericsson Review*, 3(9296):89102, 2007.

[54] Fabio Casati, Ski Ilnicki, LiJie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and dynamic service composition in eflow. In *International Conference on Advanced Information Systems Engineering*, pages 13–31. Springer, 2000.

[55] Cesare Pautasso. Composing restful services with jopera. In *International conference on software composition*, pages 142–159. Springer, 2009.

[56] Ziyan Maraikar, Alexander Lazovik, and Farhad Arbab. Building mashups for the enterprise with sabre. *Service-Oriented Computing–ICSOC 2008*, pages 70–83, 2008.

[57] Mark Pruett. *Yahoo! pipes*. O'Reilly, 2007.

[58] Shalom Tsur, Serge Abiteboul, Rakesh Agrawal, Umeshwar Dayal, Johannes Klein, and Gerhard Weikum. Are web services the next revolution in e-commerce?(panel). In *VLDB*, pages 614–617, 2001.

[59] Brahim Medjahed, Boualem Benatallah, Athman Bouguettaya, Anne HH Ngu, and Ahmed K Elmagarmid. Business-to-business interactions: issues and enabling technologies. *The VLDB JournalThe International Journal on Very Large Data Bases*, 12(1):59–85, 2003.

[60] Martin Garriga, Cristian Mateos, Andres Flores, Alejandra Cechich, and Alejandro Zunino. Restful service composition at a glance: A survey. *Journal of Network and Computer Applications*, 60:32–53, 2016.

[61] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[62] Alistair Barros, Marlon Dumas, and Phillipa Oaks. Standards for web service choreography and orchestration: Status and perspectives. In *International Conference on Business Process Management*, pages 61–74. Springer, 2005.

[63] Remco Dijkman and Marlon Dumas. Service-oriented design: A multi-viewpoint approach. *International journal of cooperative information systems*, 13(04):337–368, 2004.

[64] Valérie Issarny, Nikolaos Georgantas, Sara Hachem, Apostolos Zarras, Panos Vassiliadist, Marco Autili, Marco Aurélio Gerosa, and Amira Ben Hamida. Service-oriented middleware for the future internet: state of the art and research directions. *Journal of Internet Services and Applications*, 2(1):23–45, 2011.

[65] Johannes Zaha, Alistair Barros, Marlon Dumas, and Arthur ter Hofstede. Lets dance: A language for service behavior modeling. *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, pages 145–162, 2006.

[66] Cai Chao and Qiu Zongyan. An approach to check choreography with channel passing in ws-cdl. In *Web Services, 2008. ICWS'08. IEEE International Conference on*, pages 700–707. IEEE, 2008.

[67] Jing Li, Jifeng He, Huibiao Zhu, and Geguang Pu. Modeling and verifying web services choreography using process algebra. In *Software Engineering Workshop, 2007. SEW 2007. 31st IEEE*, pages 256–268. IEEE, 2007.

[68] Hongli Yang, Xiangpeng Zhao, Chao Cai, and Zongyan Qiu. Model-checking of web services choreography. In *Service-Oriented System Engineering, 2008. SOSE'08. IEEE International Symposium on*, pages 79–84. IEEE, 2008.

[69] Qi Yu, Xumin Liu, Athman Bouguettaya, and Brahim Medjahed. Deploying and managing web services: issues, solutions, and directions. *The VLDB JournalThe International Journal on Very Large Data Bases*, 17(3):537–572, 2008.

[70] Antonio Jorge Silva Cardoso. *Quality of service and semantic composition of workflows*. 2002.

[71] Boualem Benatallah, Quan Z Sheng, and Marlon Dumas. The self-serv environment for web services composition. *IEEE internet computing*, 7(1):40–48, 2003.

[72] Nikola Milanovic and Miroslaw Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51–59, 2004.

[73] Paweł Stelmach. Service composition scenarios in the internet of things paradigm. In *Doctoral Conference on Computing, Electrical and Industrial Systems*, pages 53–60. Springer, 2013.

[74] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[75] Dieter Uckelmann, Mark Harrison, and Florian Michahelles. An architectural approach towards the future internet of things. In *Architecting the internet of things*, pages 1–24. Springer, 2011.

[76] Bryan L Gorman, DR Resseguie, and Christopher Tomkins-Tinch. Sensorpedia: Information sharing across incompatible sensor systems. In *Collaborative Technologies and Systems, 2009. CTS'09. International Symposium on*, pages 448–454. IEEE, 2009.

[77] Nils Glombitza, Dennis Pfisterer, and Stefan Fischer. Using state machines for a model driven development of web service-based sensor network applications. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*, pages 2–7. ACM, 2010.

[78] John Soldatos, Nikos Kefalakis, Manfred Hauswirth, Martin Serrano, Jean-Paul Calbimonte, Mehdi Riahi, Karl Aberer, Prem Prakash Jayaraman, Arkady Zaslavsky, Ivana Podnar Žarko, et al. Openiot: Open source internet-of-things in the cloud. In *Interoperability and open-source solutions for the internet of things*, pages 13–25. Springer, 2015.

[79] Jean-Paul Calbimonte, Sofiane Sarni, Julien Eberle, and Karl Aberer. Xgsn: An open-source semantic sensing middleware for the web of things. In *TC/SSN@ ISWC*, pages 51–66, 2014.

[80] Danh Le Phuoc. Sensormasher-publishing and building mashup of sensor data. In *I-SEMANTICS*. Citeseer, 2009.

[81] Michael Blackstock and Rodger Lea. Iot mashups with the wotkit. In *Internet of Things (IOT), 2012 3rd International Conference on the*, pages 159–166. IEEE, 2012.

[82] Riccardo Petrolo, Aikaterini Roukounaki, Valeria Loscri, Nathalie Mitton, and John Soldatos. Connecting physical things to a smartcity-os. In *Sensing, Communication and Networking (SECON Workshops), 2016 IEEE International Conference on*, pages 1–6. IEEE, 2016.

[83] Aikaterini Roukounaki, John Soldatos, Riccardo Petrolo, Valeria Loscri, Nathalie Mitton, and Martin Serrano. Visual development environment for semantically interoperable smart cities applications. In *Internet of Things. IoT Infrastructures: Second International Summit, IoT 360 2015, Rome, Italy, October 27-29, 2015, Revised Selected Papers, Part II*, pages 439–449. Springer, 2016.

[84] Aqeel Kazmi, Zeeshan Jan, Achille Zappa, and Martin Serrano. Overcoming the heterogeneity in the internet of things for smart cities. In *International Workshop on Interoperability and Open-Source Solutions*, pages 20–35. Springer, 2016.

[85] ISO Standard. Iso 11898, 1993. *Road vehicles–interchange of digital information–Controller Area Network (CAN) for high-speed communication*, 1993.

[86] Mohammad Farsi, Karl Ratcliff, and Manuel Barbosa. An overview of controller area network. *Computing & Control Engineering Journal*, 10(3):113–120, 1999.

[87] K. McCloghrie. Management information base for network management of tcp/ip-based internets: Mib-ii, 1991.

[88] J. Case. A simple network management protocol (snmp), 1989.

[89] CCITT Recommendation. Specification of abstract syntax notation one (asn. 1), 1988.

[90] Maximilian Koegel, Markus Herrmannsdoerfer, Yang Li, Jonas Helming, and Joern David. Comparing state-and operation-based change tracking on models. In *Enterprise Distributed Object Computing Conference (EDOC), 2010 14th IEEE International*, pages 163–172. IEEE, 2010.

[91] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. Lee & Seshia, 2011.

[92] Leslie Lamport. Computer science and state machines. In *Concurrency, Compositionality, and Correctness*, pages 60–65. Springer, 2010.

[93] David J Comer. Digital logic and state machine design. 1995.

[94] Ivan Zuzak, Ivan Budiselic, and Goran Delac. A finite-state machine approach for modeling and analyzing restful systems. *Journal of Web Engineering*, 10(4):353, 2011.

[95] Jacob Beard. State machines as a service. *on Engineering Interactive Computer Systems with SCXML*, page 17, 2012.

[96] OGC. Sensor model language, 2012.

[97] W3C. State chart xml (scxml): State machine notation for control abstraction, 2015.

[98] Jonathan Marsh, David Orchard, and Daniel Veillard. Xml inclusions (xinclude) version 1.0. *W3C Working Draft*, 10, 2006.

[99] Bran Selic. Using uml for modeling complex real-time systems. In *Languages, compilers, and tools for embedded systems*, pages 250–260. Springer, 1998.

[100] JC Jensen, EA Lee, and SA Seshia. An introductory lab in embedded and cyber-physical systems. *LeeSeshia. org, Berkeley, CA*, 2012.

[101] Mudasser Iqbal, Daiqin Yang, Talha Obaid, Teng Jie Ng, and Hock Beng Lim. Demo abstract: A service-oriented application programming interface for sensor network virtualization. In *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, pages 143–144. IEEE, 2011.

[102] Hock Beng Lim, Mudasser Iqbal, and Teng Jie Ng. A virtualization framework for heterogeneous sensor network platforms. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 319–320. ACM, 2009.

[103] Nathalie Mitton, Symeon Papavassiliou, Antonio Puliafito, and Kishor S Trivedi. Combining cloud and sensors in a smart city environment. *EURASIP journal on Wireless Communications and Networking*, 2012(1):247, 2012.

[104] E Michael Maximilien, Hernan Wilkinson, Nirmit Desai, and Stefan Tai. A domain-specific language for web apis and services mashups. In *International Conference on Service-Oriented Computing*, pages 13–26. Springer, 2007.

[105] Nanalyze. Fog computing vs. cloud computing vs. edge computing, 2016.

[106] Rick Rabiser, Reinhard Wolfinger, and Paul Grunbacher. Three-level customization of software products using a product line approach. In *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, pages 1–10. IEEE, 2009.

[107] Andrew Clinick. Creating customizable web services, 2001.

[108] Linda Peters and Hasannudin Saidin. It and the mass customization of services: the challenge of implementation. *International Journal of Information Management*, 20(2):103–119, 2000.

[109] Ashraf A Shahin. Variability modeling for customizable saas applications. *arXiv preprint arXiv:1409.2156*, 2014.

[110] Arun Sharma, Rajesh Kumar, and PS Grover. A critical survey of reusability aspects for component-based systems. *World academy of science, Engineering and Technology*, 19:411–415, 2007.

[111] William Frakes and Carol Terry. Software reuse: metrics and models. *ACM Computing Surveys (CSUR)*, 28(2):415–435, 1996.

[112] James M Bieman. Deriving measures of software reuse in object oriented systems. In *Formal Aspects of Measurement*, pages 63–83. Springer, 1991.

[113] Santhi Karunanithi and James M Bieman. Candidate reuse metrics for object oriented and ada software. In *Software Metrics Symposium, 1993. Proceedings., First International*, pages 120–128. IEEE, 1993.

[114] William B. Frakes and Sadahiro Isoda. Success factors of systematic reuse. *IEEE software*, 11(5):14–19, 1994.

[115] General Electric Company, Jim A McCall, Paul K Richards, and Gene F Walters. *Factors in software quality: Final report*. Information Systems Programs, General Electric Company, 1977.

[116] M Todd Gamble and Rose Gamble. Monoliths to mashups: Increasing opportunistic assets. *IEEE software*, 25(6), 2008.

[117] Wilhelm Hasselbring and Guido Steinacker. Microservice architectures for scalability, agility and reliability in e-commerce. In *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*, pages 243–246. IEEE, 2017.

[118] Robert Lewis Biddle and Ewan D Tempero. Understanding the impact of language features on reusability. In *Software Reuse, 1996., Proceedings Fourth International Conference on*, pages 52–61. IEEE, 1996.

[119] Yida Mao, Houari A Sahraoui, and Hakim Lounis. Reusability hypothesis verification using machine learning techniques: a case study. In *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*, pages 84–93. IEEE, 1998.

[120] Paul Clements and Linda Northrop. *Software product lines*. Addison-Wesley,, 2002.

[121] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[122] Harry M Sneed. Measuring web service interfaces. In *Web Systems Evolution (WSE), 2010 12th IEEE International Symposium on*, pages 111–115. IEEE, 2010.

[123] Si Won Choi and Soo Dong Kim. A quality model for evaluating reusability of services in soa. In *E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services, 2008 10th IEEE Conference on*, pages 293–298. IEEE, 2008.

[124] K Breitfelder and D Messina. The authoritative dictionary of ieee standards terms. *Institute of Electrical and Electronics Engineers (IEEE)*, 2000.

[125] US DoD. Dod dictionary of military and associated terms. *Online*, 14:1–02, 2010.

[126] Hans van der Veer and Anthony Wiles. Achieving technical interoperability. *European Telecommunications Standards Institute*, 2008.

[127] Grace A Lewis and Lutz Wrage. Model problems in technologies for interoperability: Model-driven architecture. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2005.

[128] Luis Guijarro. Semantic interoperability in egovernment initiatives. *Computer Standards & Interfaces*, 31(1):174–180, 2009.

[129] Thomas Erl. *Soa: principles of service design*, volume 1. Prentice Hall Upper Saddle River, 2008.

[130] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Sensing as a service model for smart cities supported by internet of things. *Transactions on Emerging Telecommunications Technologies*, 25(1):81–93, 2014.

[131] Jiong Jin, Jayavardhana Gubbi, Slaven Marusic, and Marimuthu Palaniswami. An information framework for creating a smart city through internet of things. *IEEE Internet of Things Journal*, 1(2):112–121, 2014.

[132] Alexander Gluhak, Martin Bauer, Frederic Montagut, Vlad Stirbu, Mattias Johansson, Jesus Bernat Vercher, and Mirko Presser. Towards an architecture for a real world internet. In *Future internet assembly*, pages 313–324, 2009.

[133] Ruowei Xiao, Zhanwei Wu, and Kazunori Sugiura. A semantic html based approach for geosensor media. *GeoInformatica*, pages 1–22, 2016.

[134] Mark Lanus, Liang Yin, and Kishor S Trivedi. Hierarchical composition and aggregation of state-based availability and performability models. *IEEE Transactions on Reliability*, 52(1):44–52, 2003.

Appendix

A Sensor StateML Description Sample

```
<?xml version="1.0" encoding="UTF-8"?>

<stateml:StateMachine>
  xmlns:stateml="http://www2.kmd.keio.ac.jp/~ruowei.xiao/stateml" <!--
  StateML name space -->
  xmlns:scxml="http://www.w3.org/2005/07/scxml"
  xmlns:sml="http://www.opengis.net/sensorml/2.0"
  xmlns:swe="http://www.opengis.net/swe/2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xsi:schemaLocation="http://www.opengis.net/sensorml/2.0
  http://schemas.opengis.net/sensorml/2.0/sensorML.xsd"

<sml:PhysicalComponent gml:id="SENSOR_SAMPLE">

<!-- System Description -->
  <gml:description>Integrated sensor sample for environmental monitoring
  </gml:description>
  <gml:identifier codeSpace="uid">Yokohama:Hiyoshi:Kyoseikan:3FS01:01
  </gml:identifier>
  <sml:position>
      <!-- EPSG 4326 is for latitude-longitude, in that order -->
      <gml:Point gml:id="sensorLocation" srsName="http://www.opengis
      .net/def/crs/EPSG/0/4326">
         <gml:coordinates>35.554498 139.6485728</gml:coordinates>
      </gml:Point>
  </sml:position>

<!-- Device Capabilities -->
```

```
  <sml:capabilities name="specifications">
      <sml:CapabilityList>
        <sml:capability name="measurementProperties">
            <swe:DataRecord definition="http://sensorml.com/ont/swe/
                property/MeasurementProperties">
                <swe:field name="outputInterval">
                        <swe:Quantity definition="http://sensorml.com/
                        ont/swe/property/OuputPeriod">
                        <swe:uom code="s" />
                        <swe:value>2.58</swe:value>
                        </swe:Quantity>
                </swe:field>
                        <swe:field name="SampleInterval">
                        <swe:Quantity definition="http://sensorml.com/
                        ont/swe/property/SamplePeriod">
                        <swe:uom code="s" />
                        <swe:value>300</swe:value>
                        </swe:Quantity>
                </swe:field>
                </swe:DataRecord>
        </sml:capability>
          </sml:CapabilityList>
  </sml:capabilities>

<!-- Inputs = Observed Properties -->
  <sml:inputs>
      <sml:InputList>
        <sml:input name="temperature">
            <sml:ObservableProperty definition="http://sweet.jpl.nasa
            .gov/2.3/propTemperature.owl#Temperature"/>
            </sml:input>
            <sml:input name="Air">
                <sml:ObservableProperty definition="http://ontology
                .example.org/phenomenon/air"/>
            </sml:input>
                    <sml:input name="Sound">
                <sml:ObservableProperty definition="http://ontology
                .example.org/phenomenon/sound"/>
            </sml:input>
```

```
      </sml:InputList>
  </sml:inputs>

<!-- Observed Property = Output -->
  <sml:outputs>
      <sml:OutputList>
        <sml:output name="IntegratedSensorStream">
            <sml:DataInterface>
                <sml:data>
                    <swe:DataStream>
                        <swe:elementType name="environmental_data">
                            <swe:DataRecord>
                                <swe:field name="Relative Humidity">
                                    <swe:Quantity definition="http://
                                    mmisw.org/ont/CUAHSI/Atmospheric
                                    HydrologicCore/relativeHumidity">
                                        <swe:uom code="%RH" />
                                    </swe:Quantity>
                                </swe:field>
                                <swe:field name="Temperature">
                                    <swe:Quantity definition="http://
                                    mmisw.org/ont/cf/parameter/
                                    air_temperature">
                                        <swe:uom code="Cel"/>
                                    </swe:Quantity>
                                </swe:field>
                                <swe:field name="Pressure">
                                    <swe:Quantity definition="http://
                                    mmisw.org/ont/cf/parameter/
                                    barometric_pressure">
                                        <swe:uom code="hPa" />
                                    </swe:Quantity>
                                </swe:field>
                                <swe:field name="Sound Intensity">
                                    <swe:Quantity definition="http://
                                    mmisw.org/ont/cf/parameter/
                                    sound_intensity_level_in_air">
                                        <swe:uom code="dB" />
                                    </swe:Quantity>
```

```
                                </swe:field>
                                <swe:field name="Battery voltage">
                                    <swe:Quantity definition="http://
                                    sensorml.com/ont/swe/property/
                                    Voltage">
                                        <swe:uom code="mV" />
                                    </swe:Quantity>
                                </swe:field>
                            </swe:DataRecord>
                        </swe:elementType>

                        <swe:encoding>
                            <swe: TextEncoding tokenSeperator=","
                            blockSeparator=" "/>
                        </swe:encoding>

    <!-- a Real-Time-Protocol (RTP) server that continues to stream real
    time measurements -->
                        <swe:values xlink:href="rtp://myServer.com:4563/
                                                sensor/02080"/>
                    </swe:DataStream>
                </sml:data>
            </sml:DataInterface>
        </sml:output>
    </sml:OutputList>
  </sml:outputs>
</sml:PhysicalComponent>

<!-- Sensor as a State Machine Description -->
<!-- We dont specify internal events that trigger state to naturally
transfer, e,g, when sensing state is over,
     it will continue to outputting state without user intervention. -->
<!-- Events that trigger state transition are always binded to
significant variables and their value changes.
     And the variables must be described in the SensorML in order to
     state their range and I/O type. -->
  <stateml:states>
    <scxml:state id="SensorSample" initial="idle">
        <scxml:state id="idle">
```

```
            <scxml:transition event="Battery voltage: U_INT_8"
            type="internal" target="sleep"></scxml:transition>
            <scxml:transition event="outputInterval: U_INT_8"
            type="external" target="settingOutputInterval">
            </scxml:transition>
            <scxml:transition event="sampleInterval: U_INT_8"
            type="external" target="settngSampleInterval">
            </scxml:transition>
        </scxml:state>
        <scxml:state id="settingSampleInterval">
            <scxml:transition target="idle"></scxml:transition>
        </scxml:state>
        <scxml:state id="settingOutputInterval">
            <scxml:transition target="idle"></scxml:transition>
        </scxml:state>
        <scxml:state id="sleep"></scxml:state>

        <scxml:state id="output">
            <scxml:transition event="Temperature: S_INT_16; Relative
            Humidity: S_INT_16; Pressure: S_INT_16; Sound: S_INT_16"
            type="internal" target="output">
            </scxml:transition>
                </scxml:state>
    </scxml:state>
  </stateml:states>
</stateml:StateMachine>
```

B Actuator StateML Description Sample

```
<?xml version="1.0" encoding="UTF-8"?>

<stateml:StateMachine>
  xmlns:stateml="http://www2.kmd.keio.ac.jp/~ruowei.xiao/stateml"
  <!-- StateML name space -->
  xmlns:scxml="http://www.w3.org/2005/07/scxml"
  xmlns:sml="http://www.opengis.net/sensorml/2.0"
  xmlns:swe="http://www.opengis.net/swe/2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xsi:schemaLocation="http://www.opengis.net/sensorml/2.0
  http://schemas.opengis.net/sensorml/2.0/sensorML.xsd"

<sml:PhysicalComponent gml:id="ACTUATOR_SAMPLE">

<!-- System Description -->
  <gml:description>Robot Arm: RA1-PRO</gml:description>
  <gml:identifier codeSpace="uid">Yokohama:Hiyoshi:Kyoseikan:ProjectRoom
  </gml:identifier>

<!-- metadata deleted for brevity sake -->

<!-- Setting Parameters = Input -->
<sml:inputs>
    <sml:InputList>
        <sml:input name="servos">
            <swe:field name="mode">
                <swe:Quantity definition="http://kmd.keio.co.jp/robotArm
                /property/servoMode">
                    <swe:value>FKMode</swe:value>
            </swe:Quantity>
            </swe:field>
            <swe:field name="angle1">
                <swe:Quantity definition="http://kmd.keio.co.jp/robotArm
                /property/servoAngle">
                    <swe:value>0</swe:value>
                </swe:Quantity>
            </swe:field>
            <swe:field name="angle2">
                <swe:Quantity definition="http://kmd.keio.co.jp/robotArm
                /property/servoAngle">
                    <swe:value>0</swe:value>
                </swe:Quantity>
            </swe:field>
            <swe:field name="angle3">
                <swe:Quantity definition="http://kmd.keio.co.jp/robotArm
                /property/servoAngle">
                    <swe:value>0</swe:value>
```

```
                </swe:Quantity>
            </swe:field>
             <swe:field name="angle4">
                <swe:Quantity definition="http://kmd.keio.co.jp/robotArm
                /property/servoAngle">
                    <swe:value>0</swe:value>
                </swe:Quantity>
            </swe:field>
        </sml:input>
        <sml:input name="coordinates">
            <swe:field name="x">
                <swe:Quantity
                definition="http://kmd.keio.co.jp/robotArm/property/x">
                    <swe:value>0</swe:value>
                </swe:Quantity>
            </swe:field>
            <swe:field name="y">
                <swe:Quantity
                definition="http://kmd.keio.co.jp/robotArm/property/y">
                    <swe:value>0</swe:value>
                </swe:Quantity>
            </swe:field>
            <swe:field name="z">
                <swe:Quantity
                definition="http://kmd.keio.co.jp/robotArm/property/z">
                    <swe:value>0</swe:value>
                </swe:Quantity>
            </swe:field>
        </sml:input>
    </sml:InputList>
  </sml:inputs>

</sml:PhysicalComponent>

<!-- Actuator as a State Machine Description -->
<stateml:states>
  <scxml:state id="robotArm-RA1-PRO" initial="on">
    <scxml:state id="on">
           <scxml:transition type="internal" target="settingFKMode">
```

```
        </scxml:transition>
    </scxml:state>
    <scxml:state id="settingFKMode">
            <scxml:transition type="internal" target="FKMode">
            </scxml:transition>
            <scxml:transition type="external" event="mode: STRING"
            target="settingIKMode"></scxml:transition>
    </scxml:state>
    <scxml:state id="settingIKMode">
            <scxml:transition type="internal" target="IKModeIdle">
            </scxml:transition>
            <scxml:transition type="external" event="mode: STRING"
            target="settingFKMode"></scxml:transition>
    </scxml:state>
    <scxml:state id=" FKMode">
        <scxml:state id="FKIdle">
                <scxml:transition type="external" event="mode:
                STRING" target="settingIKMode"></scxml:transition>
                <scxml:transition type="external" event="angle1:
                S_INT_16; angle2: S_INT_16;angle3: S_INT_16; angle4:
                S_INT_16" target="FKMotion"></scxml:transition>
        </scxml:state>
        <scxml:state id="FKMotion">
                <scxml:transition type="internal" target="FKIdle">
                </scxml:transition>
        </scxml:state>
    </scxml:state>
    <scxml:state id=" IKMode">
        <scxml:state id="IKIdle">
                <scxml:transition type="external" event="mode:
                STRING" target="settingFKMode"></scxml:transition>
                <scxml:transition type="external" event="coordinateX:
                S_INT_16; coordinateY: S_INT_16; coordinateZ:
                S_INT_16" target="IKMotion">
                </scxml:transition>
        </scxml:state>
        <scxml:state id="IKMotion">
                <scxml:transition type="internal" target="IKIdle">
                </scxml:transition>
```

```
        </scxml:state>
    </scxml:state>
 </scxml:state>
</stateml:states>

</hsml:StateMachine>
```

C Servitization Example in Node.JS

```
// drone_server.js
// This is a sample server-end script by NodeJS to host a drone.
// It gave an example that how to wrap an actuator's orginal API
// into standard FSM interface, which can be exported as a module.

var express = require('express')
    ,cors = require('cors');
var app = express();
var port = 1337;
var arDrone = require('ar-drone');
var client  = arDrone.createClient();
var autonomy = require('ardrone-autonomy');
var bodyParser = require('body-parser');
var currentState = 'off';


app.use(cors());
app.use(bodyParser.urlencoded({extended: false}));
app.use(bodyParser.json());

var NodeServerArDrone = function() {

    app.get('/', function(request, response){
        response.status(200).json({state machine:{...}});
        //return scxml or other state machine description here
    });

    app.get('/state', function(request, response){
        response.status(200).json({state:currentState});
```

```
});

app.post('/takeoff', function(request, response){
    console.log("Taking off...");
    currentState = 'takeoff';
    client.ftrim();
    client.takeoff();
    currentState = 'hovering';
    response.status(200).json({state:currentState});
});

app.post('/land', function(request, response){
    console.log("Stopping activities and landing...");
    client.stop();
    client.land();
    currentState = 'land';
    response.status(200).json({state:currentState});
});

app.post('/flying', function(request, response){
    var speed = request.body.speed;
    console.info("speed:",speed);
    if(currentState == 'land')
        client.takeoff();
    if(speed > 0)
    {
        client.up(speed);
    }
    else if (speed < 0)
    {
        client.down(Math.abs(speed));
    }
    currentState = 'flying';
    response.status(200).json({state:currentState});
});

app.post('/hovering', function(request, response){
    client.stop();
    currentState = 'hovering';
```

```
        response.status(200).json({state:currentState});
    });

    app.listen(port);
    console.log('Node.js express server started on port %s', port);
};

module.exports = NodeServerArDrone;
```

D User Experiment Guidance

D.1 Experiment Tasks

1. Introduce a temperature sensor that connected to an Arduino into target system and show the data reading.
2. Use sensor data to trigger an actuator, i.e. a LED in this case, to complete an automation.

D.2 Experiment Environment

Hardware

1. MSI GS60 notebook (Intel i7-6700HQ CPU @ 2.60GHz, RAM 16.0G)
2. Arduino Uno * 1 (connected to 1 via COM3)
3. LM35 Temperature Sensor * 1(connected to 2 via Analog pin 0)
4. LED * 1 (connected to 2 via Digital pin 11)

Software

1. System: Windows 10 64bit
2. Web Browser: Chrome
3. Code Editor: Brackets
4. Local Server Environment: XAMPP

D.3 Comparison Systems

1. Home Assistant:`https://home-assistant.io`
2. Node Red:`https://nodered.org`
3. PubNub+Eon:`https://www.pubnub.com`
4. HSML:`http://www2.kmd.keio.ac.jp/~ruowei.xiao/hsml`

D.4 Experiment Procedure

Home Assistant

1. Start Home Assistant using command mode by inputting:
   `py -m homeassistant --open-ui`
2. Open homeassistant configuration file using Bracket or any code editor at
   `C:\Users\guest\AppData\Roaming\.homeassistant\configuration.yaml`
   *Participants need to repeat 1.1 to restart the tool each time after editing the file.
3. Add Arduino by referring to:
   `https://home-assistant.io/components/arduino/`
   *The serial port name given in the reference is in Linux system. Participants may need to get corresponding serial port name in Windows by themselves.
4. Add temperature sensor and record current data value (on the top of web-page) by refering to:
   `https://home-assistant.io/components/sensor.arduino/`
5. Add LED light by referring to:
   `https://home-assistant.io/components/switch.arduino/`
6. Switch on the LED light when sensor reading over 60. Use configuration -⟩ automation to set up automation rules. Participants can refer to to the instructions in the same page.

Node Red

1. Start node-red in command mode by inputting:
   `E:`
   `cd node\node_modules\node-red`
   `node red.js`
   And use web browser to access `http://127.0.0.1:1880/`

2. Drag and drop an Arduino input widget, double click to configure the sensor pin.

3. Drag and drop a debug output widget, connect it with arduino sensor input and deploy. See the sensor reading result in debug window.

4. Drag and drop an Arduino output widget, double clikc to configure the LED pin.

5. Drag and drop a function widget, and connect Arduino sensor input with Arduino LED ouput via function and deploy. The purpose of function is to automatically switch the LED on when the sensor reading over 60. Reference:

```
var temp = msg.payload;
var led = 0;
if(temp >= 60){led=1;}
var msg = {payload: led};
return msg;
```

PubNub Eon

1. Open pubnub website at `www.pubnub.com`register and obtain pub/sub keys.

2. Refer to the JavaScript program template at `E:\node\PubNub\server.js`, and edit it using Brackets to get sensor data from Arduino and send to the pubnub server:

```
var five = require("johnny-five");
var board = new five.Board();
var PubNub = require("pubnub");
var reading, message;
```

```
board.on("ready", function() {
var sensor = new five.Sensor({
            pin: "SENSOR_PIN_NUMBER_HERE",
            freq: 1000
  });
    var pubnub = new PubNub({
    publishKey : 'YOUR_PUBKEY_HERE',
    subscribeKey : 'YOUR_SUBKEY_HERE'
  });
sensor.on("data", function() {
            reading = this.value*5*1000/1024/10;
            reading = reading.toFixed(1);
            pubnub.publish({
                    channel   : 'pubnub-eon-iot',
                    message   : {
                            eon:{'Temperature': reading}
                            }
                    });
            });
    });
```

3. Start running server.js in command mode by inputting:
   `E:`
   `cd node\hsml`
   `node server.js`

4. Edit the JavaScript template at `D:\xampp\htdocs\eon\Index.html` and run it on local web server to subscribe pubnub data and visualize:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title></title>
<script src="lib/pubnub.js"></script>
<script type="text/javascript" src="https://pubnub.github.
```

```
io/eon/v/eon/1.0.0/eon.js">
</script>
<link type="text/css" rel="stylesheet" href="https://
pubnub.github.io/eon/v/eon/1.0.0/eon.css"/>
</head>

<body>
    <div id="chart"></div>
</body>

<script>
pubnub = new PubNub({
    publishKey : ' YOUR\_PUBKEY\_HERE ',
    subscribeKey : ' YOUR\_SUBKEY\_HERE '
});

    eon.chart({
    channels: ["pubnub-eon-iot"],
        history: true,
        flow: true,
            pubnub: pubnub,
            generate: {
                    bindto: '#chart',
                    data: {
                            labels: true
                    }
            }
    });
</script>
</html>
```

5. Run xampp server and use web browser to visit `localhost/eon` to view visualization result.

6. Change visualization effect to bar diagram referring to:
`https://www.pubnub.com/developers/eon/chart/bar/`

HSML

1. Start HSML server in command mode by inputing:
   `E:`
   `cd node\hsml`
   `node server.js`
   Access in web browser: `http://localhost:6147/`

2. Obtain necessary information regarding resource addresses and controllable states, by accessing FSM descriptions of sensor and LED as shown in Figure D.1 and D.2:

3. Visit `localhost/hsml` using web browser and input HSML in text editor. Use resource descriptor $< loc >$ to introduce sensor, and see data diagram at visualizer. HSML grammar:

   ```
   <loc id="SENSOR_ID(combination of 26 characters & number)"
   src="SENSOR_URI" x="POSITION_X_COORDINATE(INT 0~100)"
   y="POSITION_Y_COORDINATE(INT 0~100)"
   viz="VISUALIZATION_EFFECT(bar,area,line)"
   type="DEVICE_TYPE(sensor,actuator)"></loc>
   ```

   According to the information provided in the Figure D.1, participants are supposed to fill in correct URI like:

   ```
   <loc id="tempSensor" src="http://131.113.136.95:6147/
   tempSensor/state/output" x="10" y="10"
   type="sensor" viz="bar"></loc>
   ```

4. Using same grammar to create an LED descriptor. And according to Figure D.2, participants are expected to complete the following HSML text like:

   ```
   <loc id="ledActuator" src="http://131.113.136.95:6147/
   ledActuator/state" x="20" y="10"
   type="actuator"></loc>
   ```

5. Using state transfer descriptor $< lnk >$ to mapping sensors temperature(0˜100) reading to LEDs brightness(0˜255), and make LED growing brighter along with sensors reading rising.HSML grammar:

```
<lnk function="LINEAR(SENSOR_ID.temperature, LED_ID.
brightness, RATIO(int));"></lnk>
```

And participants are expected to fill in the HSML text like:

```
<lnk function="LINEAR(tempSensor.temperature, ledActuator.
brightness, 2.55);"></lnk>
```

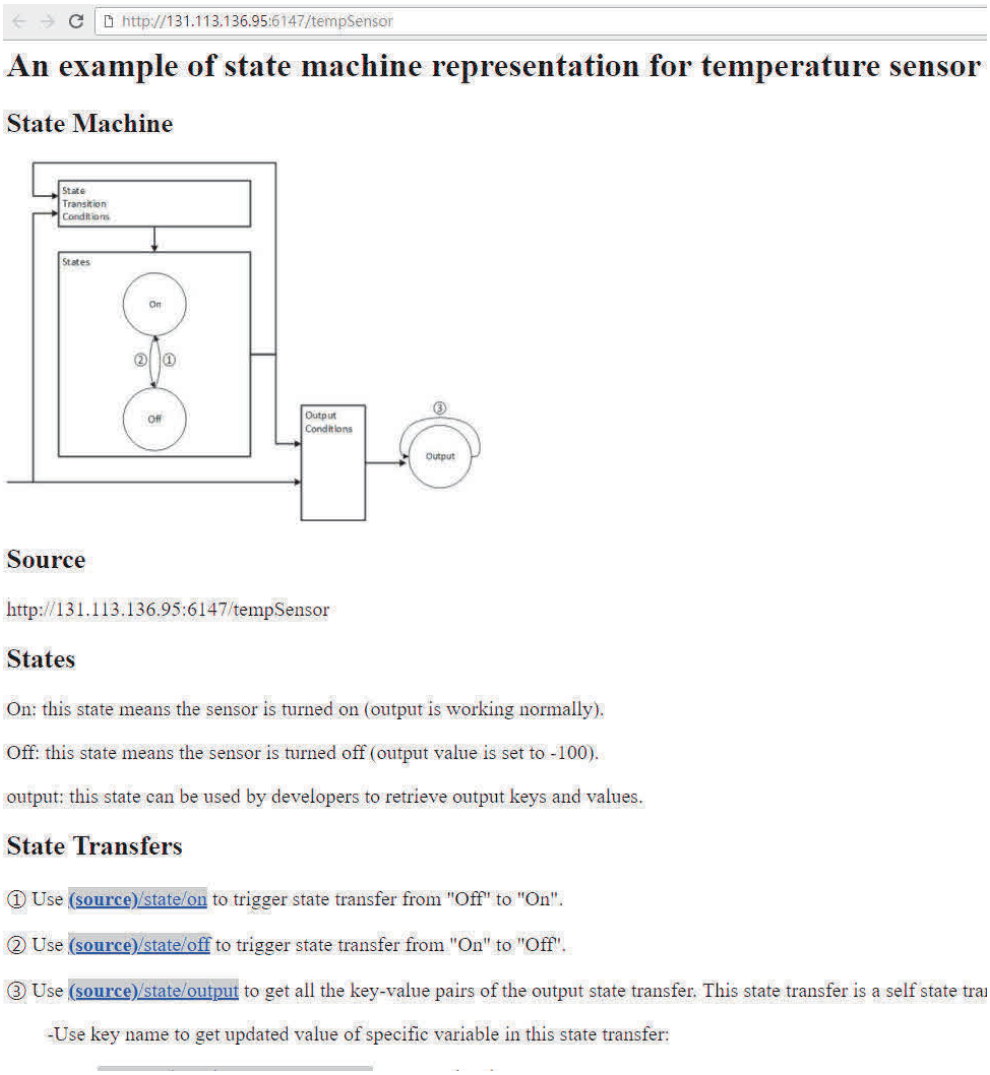


Figure D.1: Graphic FSM Service Description for Temperature Sensor

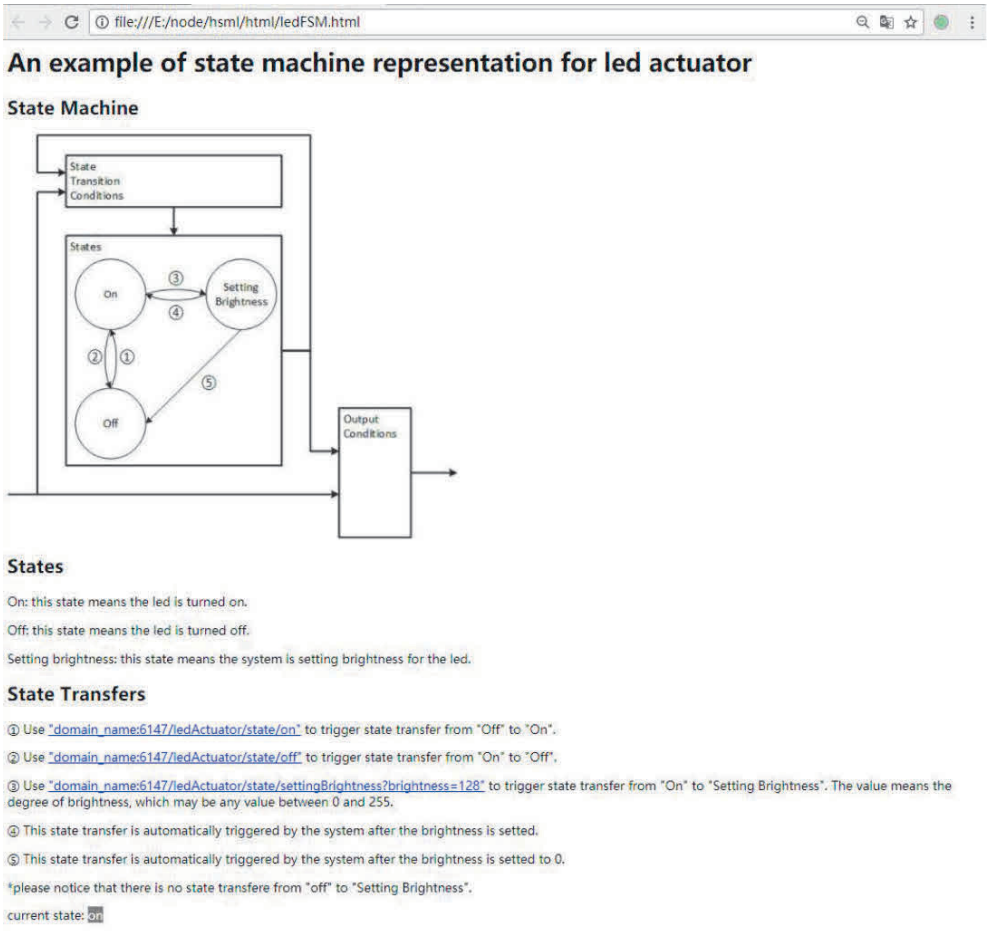


Figure D.2: Graphic FSM Service Description for LED