Design and Implementation of a Unified Programming Framework for Things, Web and Cloud

Debashish Ghosh

School of Computer Science
McGill University
Montreal, Canada

October 2014

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Master of Science.

Dedication

To Mom, Dad and the Internet of Things community.

Acknowledgements

I would like to thank my thesis supervisor, Professor Muthucumaru Maheswaran, for being a source of inspiration and providing a continuous influx of new ideas and concepts that could make a difference to the world of Internet of Things and technology in general. I would also like to acknowledge other students in the Advanced Network Research Lab. Fan Jin, for always being there and helping out in working with arduino processors. My friends Syed Ahmed, Vaibhav Somani, Sridipta Misra, Bhaskar Pilania, Mohit Shah, Sujay Kathrotia and Sameer Jagdale for always supporting. Lineker Tomazeli for helping out with the ideas about SpaceOS. Robert Wenger for the work on the video server and converting the server from Twisted to Tornado. Julien Lord for providing hlib.

Abstract

Internet of Things (IoT) are quite diverse in their capabilities: ranging from tiny sensors to Internet connected appliances. As a result, a particular computing activity might be split across many elements from things, web, and cloud. Therefore, a programming framework targeted towards IoT must be quite flexible in allowing the developer to partition the processing actions among the participating elements without requiring the developer to strictly adhere to predefined service interfaces. The programming framework should keep the language familiar and have minimal learning beyond existing languages and technologies. In this thesis, I present JavaScript Arduino Development Environment (JADE), a framework that allows a developer to mix C and JavaScript constructs with JADE keywords to construct a complete program to solve a particular computing activity. The combination of C, the most commonly used language for the things and JavaScript the most rapidly expanding and extensively used language of the web, could prove to be extremely potent. I describe the language constructs introduced by JADE and explain how they can be used to implement different software interaction patterns among thing, web, and cloud. The next expected frontier in the evolution of the Internet is the subsumption of physical objects and the spatial interactions with them into the internet. This leads to another important feature of JADE, which is to leverage the facilities provided by SpaceOS, a system software stack for smart computing environments; to provide a simpler programming model for the things. I implemented a proof-of-concept prototype of JADE over Intel Galileos, web, and cloud. The results from the experiments are described in the chapter on Experimental Results.

Résumé

L'Internet des Objets (IdO) est trs diverses dans son application: allant de minuscules capteurs des appareils connects Internet. Par consquent, une activit de programmation particulire pourrait tre rpartie entre de nombreux lments tels les objets, le web et le cloud. Par consquent, un cadre de programmation orient vers l'Internet des Objets doit tre assez souple pour permettre au dveloppeur de partitionner les actions de traitement entre les lments cibls sans l'obliger de se conformer strictement aux interfaces de services prdfinis. Le cadre de programmation devrait garder un langage familier et comporter peu dapprentissage au-del du langage et des technologies existantes. Dans cette thse, je prsente JavaScript Arduino Development Environment (JADE), un cadre de programmation qui permet un dveloppeur de mlanger le C et le JavaScript en combinaison avec des mots cls de JADE pour construire un programme complet capable de rsoudre une activit informatique particulire. La combinaison de C, le langage le plus couramment utilis pour les objets et de JavaScript, le langage le plus couramment utilis pour le web et qui connat une expansion rapide, pourrait s'avrer extrmement puissant. Je dcris la structure des langages introduites par JADE et explique comment ils peuvent tre utiliss pour implanter diffrents modes d'interactions logiciels parmi les objets, le web, et le cloud. La prochaine frontire prvue dans l'volution dInternet est la subsomption des objets physiques et les interactions spatiales avec ceux-ci dans sur Internet. Cela introduit une autre caractristique importante de JADE, qui consiste exploiter les installations fournies par SpaceOS, un logiciel de systme pour les environnements informatiques intelligents; fournir un modle de programmation plus simple pour les objets. J'ai mis en place un prototype de preuve de concept pour JADE sur Intel Galile, le web, et le cloud. Les rsultats des expriences sont dcrits dans la chapitre des rsultats exprimentaux.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

A recent forecast made by International Data Corporation (IDC) projected Internet of Things (IoTs) and the associated ecosystem to be an $8.9 trillion market by 2020 and include more than 200 billion connected things. To realize such a scale of expansion, it is not sufficient to just create the devices and deploy them, the associated ecosystem needs to be developed and users need to embrace IoT. Users will start embracing IoT only when compelling applications that solve real problems are available for IoT. Therefore, it is important to focus on creating programming frameworks that will allow developers to easily develop high-quality applications for IoT.

Due to various factors, programming IoT is different from normal computers. One difficulty is that 'things' may not be capable of processing a complete computing activity by themselves. They may need the computational capabilities of cloud-based backend to complete the processing tasks and a web-based frontend to interact with the user. Therefore, programming the thing can be quite complicated by the fact the programmers need to deal with disparate elements to complete an activity. Another difficulty is the mobility of ‘things’ or other factors that can make ‘things’ less available. Yet another difficulty and perhaps the hardest is the interoperability of the ‘things’.

In this thesis, we present a unified programming framework called JADE (an acronym standing for JavaScript Arduino Development Environment) for IoT. JADE is a hybrid language that uses few keywords to mix C/C++ with JavaScript. For instance, we can have JavaScript implementation of a C function. The JADE preprocessor separates the JavaScript and C/C++ code and makes the necessary linking between the corresponding functions. Using JADE, we can write a program for a ‘thing’ that partly runs on the thing and partly runs on the web or cloud. The execution of all components are coordinated by the portion of the program that runs on the ‘thing’. The C/C++ portion augmented with glue code is compiled to run on the ‘thing’. Similarly, the JavaScript portion and the corresponding glue code is injected into the web or cloud. With the rapid ascension of NodeJS in cloud computing, it is possible to have cloud-based backends running JavaScript code that is injected by the JADE program.

JADE makes it simpler to develop many IoT applications by bringing the well known “single node view” to IoT programming. It fuses the ‘thing’, web, and cloud such that cloud and web can be accessed via remote procedure calls implemented by the JADE runtime. For instance, an intelligent thermostat application can be developed in a single set of source files written in JADE. The C/C++ portion does the sensing and control part of the thermostat functions. The JavaScript portion is responsible for storing the sensed values in the clouds, running the prediction algorithms on the cloud, and realizing the user interface on the web. Using JADE, the intelligent thermostat developer would not be restricted by any preconceived service interfaces. She is free to program all elements according to the needs of the application at the same time reusing existing libraries and services.

1.2 Motivation

Programming IoT presents many unique challenges. One of the key problems for programming in IoT is their heterogeneous nature with heavily constrained sensors in one extreme and fully-featured smart devices in the other extreme. While the resource-constrained sensors can be dedicated to a specific computing activity, the fully featured smart devices could be hosting apps that allow them to perform different tasks at users' discretion. Therefore, the debate [1] [2] rages on regarding the programming models that should be supported by IoT.

There are three major ways for approaching the programmability problem in a heterogeneous distributed system such as the IoT: use service-oriented computing, develop a new programming language, and develop a library that exposes a uniform interface on all devices. The Web of Things (WoT) [3] is a variation of IoT that implements the service-oriented computing paradigm. Although service-oriented computing is quite powerful, one of the drawbacks is the requirement to use the services "as is" using the interfaces exposed by them. In service-oriented computing, each "service endpoint" is maintained by a service provider who creates and maintains the offered services. Therefore, it is difficult or impossible for another party to customize the services offered by an endpoint. As a result, with WoT, the common case is to reuse existing services through the APIs offered by them.

Over the years, many programming languages have been developed for IoT like environments [4]. Although a programming language that is designed for IoT can bring the optimal set of features, it will also have many drawbacks. The most important among them is the resistance from the developers to learn an entirely new language. Also, implementing the new language on all constituent devices is another major challenge. Due to these reasons, most proposals for new languages for heterogeneous distributed systems have remained academic curiosities.

Another powerful approach to programming IoT is to develop libraries that run on all constituent devices and support a uniform interface for communications. The AllJoyn [5] framework from Qualcomm is one such example. The biggest advantage of this approach is that the developer need not learn a new language; instead, she needs to learn the library APIs. The drawback is the effort needed in maintaining the library on all devices. Because the libraries need to be packaged with the operating system and often require hardware support, the device manufacturers need to be engaged in the development of the libraries. Another significant drawback of this approach is its incompatibility with the web. The web browsers have very restricted ways of interacting with other components. Therefore, a program running in a browser cannot utilize such libraries.

1.3 Problem Definition

The primary goal of this thesis is to develop a unified programming framework which allows the developer in the domain of IoT to write code as a single program. This should make the job of the developer a lot simpler. Secondly, there is a need to reduce the barrier for developers by reusing widely used existing technologies, rather than introducing new technologies.

In addition, to expedite the deployment of a system software stack for smart computing environment called SpaceOS, by providing a simpler programming framework for the ‘things’.

1.4 Thesis Contribution

The thesis makes the following contributions. First, it presents an architectural framework of how users and things can communicate via a server system. The thesis provides a programming framework, JADE that allows the user to write code for smart devices in the world of Internet of Things (IoT) to be executed on the ‘thing’, on the server and on the browser. There is a novel contribution in terms of interaction patters between the ‘thing’, server and the user. It offers standard options where the ‘thing’ can offer API and user can call functions of the API. Furthermore, it offers an event-based system where events can be triggered by both the user or the ‘thing’ together with a publish/subscribe system that allows to subscribe to events and reacts to them. Finally interaction can also occur directly through variables that appear to be replicated across devices and servers and cam be manipulated directly by all entities.

1.5 Thesis Organization

Chapter 2 provides the background information for the thesis. Chapter 3 describes the design rationale for a unified programming framework. Chapter 4 presents the details of the JADE programming framework. The experimental results are discussed in Chapter 5. Chapter 6 discusses some potential applications of the JADE framework in different domains. Chapter 7 discusses the related works.

Chapter 2

Background Information

2.1 Overview

The background information has been divided into two parts, the first part consists of ideas contributing to JADE and the second part comprises of ideas supported by JADE. The topics in the first part are important for helping in the understanding of JADE. Topics like RPC, RMI, CORBA, REST and SOAP discuss different methods of communication and interaction within a network or between different networks. Their understanding is important as the underneath architecture of JADE for communication between the 'things', the cloud server and the web is inspired and based on these standard protocols. AllJoyn is a peer to peer framework to help overcome interoperability issues in IoT. Tiny Web Services attempts to provide interoperability in IoT at the application level. The knowledge of these two is important as JADE is positioned as a programming framework for the diverse world of IoT, and attempts to tackle the challenge of interoperability. The second part discusses topics that JADE supports and could contribute to. Topics like Cyber Physical Systems, Ambient Intelligence and Semantic Technology have become more important with the rise of IoT. The development of JADE could assist and contribute to each of these important fields.

2.2 Ideas Contributing to JADE

2.2.1 RPC and RMI

Procedure calls are used for transfer of control over the same program. Remote Procedure Calls (RPC) [6] proposes that the same method could be used over communication networks too. In Remote Procedure Calls, upon invocation, the calling environment is suspended, and the parameters are passed to the environment where the procedure is to be executed, referred to as the callee procedure. The results are returned to the calling environment, where execution resumes as if returning from a simple single-machine call. Advantages of RPC include clean and simple semantics, efficiency and generality. RPC gained popularity as a means of making distributed computing easier, as at the time of its inception even experienced system programmers didn't have enough expertise to build distributed systems with existing tools. RPC structure is based on the concept of stubs. There are five components involved while making a remote call. These include the user, the user-stub, the RPC communication package, the server-stub, and the server. The user, the user-stub and one instance of RPCRuntime execute in the caller machine; the server, the server-stub and another instance of RPCRuntime are executed in the callee machine. The programmer also needs to consider the steps involved in inter machine binding. There are two aspects involved in binding. Firstly, how would the client of the binding mechanism determine what he wants to bind to. Secondly, how could caller determine the machine address of the callee. First problem is resolved by using a naming convention, which binds the importer of an interface to the exporter of the interface, by specifying the type and the instance of the interface. The second problem to determine the location, could be resolved either by using the network address of the machine with which they wish to communicate in the application program itself; alternatively a broadcast protocol could be used to locate a specific machine. With JDK 1.1, Java introduced the object oriented equivalent of the Remote Procedure Call called Remote Method Invocation (RMI). RMI is native to Java and depends on multiple features of Java objects like serialization, portability and Java interface definition. Tightness with Java makes it impractical to be used with applications written in other languages. RMI is a Java-to-Java technology. If we want a Java client to use RMI to communicate with a remote object in another language, it must be done using a Java intermediary that is co-located with the “foreign” remote object. So, to use RMI, a Java middleware needs to be provided. RMI [7] is much more flexible compared to Remote Procedure Call (RPC). It allows polymorphism, so remote classes could be downloaded into running application. RMI offers more options for parallel and distributed programming. Communication overhead of the Java RMI implementation remains a weakness. High latency is typical for distributed systems for which they were created, however for more tightly coupled parallel machines such latency is unacceptable.

2.2.2 AllJoyn

AllJoyn [5] is an open source Android-based peer-to-peer communication framework. It helps in overcoming interoperability issues in IoT by introducing the D-Bus protocol as an abstraction layer, that allows it to run on multiple operating systems like

Linux, Android, Microsoft Windows, iOS, OS X etc. AllJoyn uses a Java-like location transparent RMI service. The main goal of AllJoyn is to provide a software bus for distributed advertising and discovery services. Another software abstraction for bus attachment allows the peers to be connected by providing a unique well known name. A hierarchical naming system, similar to the UNIX file system, is used for arranging the bus objects. Clients should use proxy bus objects for ease of interaction. Running AllJoyn on android is enabled by using the AllJoyn daemon. The OS abstraction layer assists the daemon to be run on different operating systems. The bus attachment is a link to the IPC that connects the clients and services to the AllJoyn daemon. JADE differs from AllJoyn as it does not expose a software bus for peers to attach; rather it provides a unified programming model for the developers to write code for the thing, server or web. Functions like 'jacall' explained in Chapter 4, could be used to achieve peer to peer communication. As the developer is able to code for both the server and the web, this allows the freedom to control the API present in the server through 'jadef'.

2.2.3 CORBA

Common Object Request Broker Architecture through the use of object-oriented model allows communication between different operating systems, programming languages. The key features of CORBA includes a good service description, strong typing, ensure atomic transactions and language mapping. While some important drawbacks of CORBA include a two step activation process for the CORBA object, simulating same address space and providing state to objects. In contrast to RPC and RMI described before, CORBA is an integration technology which doesn't exist as a programming language, but rather as an Interface Definition Language (IDL); and converts code in one language through the ORB as a generalised interface, to another desired programming language.

2.2.4 SOAP

Simple Object Access Protocol (SOAP) [8] was the first divergence from the traditional RPC model. SOAP defined and improved the wire model. SOAP also defined a modular description framework called WSDL. It's different from other protocols in the sense that it has no objects, involves no directory and there is no code mobility. Although SOAP is a Remote Procedure Call, it has differences from the traditional RPC model, and therefore perceived as a divergent from the typical RPC characteristics. SOAP [9] decouples the encoding, the protocol and the transport. It uses an envelope syntax for sending and receiving XML messages between clients and services. SOAP is a protocol for decentralized and distributed system, to increase the power of internet to pass typed information between the client and services. SOAP is an XML based object invocation protocol, and was originally developed for distributed applications to communicate over HTTP. SOAP defines the use of XML and HTTP to access platform independent services. In contrast, XML-RPC is a Remote Procedure Calling protocol that works over the Internet, and is really an XML-RPC message that is an HTTP-POST request. The body of the request is in XML. A procedure executes on the server and the value it returns is also formatted in XML. The main difference between SOAP and traditional RPC lies in the fact that in SOAP we use procedures which have named parameters and order is irrelevant where as in XML-RPC order is relevant and parameters do not have names. SOAP is about document-level transfer, while traditional RPC is more about values being transferred. Another difference is that SOAP is extremely verbose, traditional RPC is simpler in comparison. JADE has some similarities to SOAP, but instead of using the envelope syntax for sending XML messages, JADE wraps up the message to be sent in a JSON object, which is parsed at the server or the browser, and the appropriate function is called. Using this approach the message sent is less convoluted and there is lesser content compared to sending it in the enveloper syntax format as used by SOAP.

2.2.5 REST

REST [10] enables component interactions in a layered client server style architecture with the added constraint of a generic resource interface, which allows inspection by intermediaries. The REST service is pull based, the consuming component, pulls representation whenever it suspects a state change to occur. Although it is less efficient than a push based system, for a single client system, but for the Web, where its not practical to have a push based system, the pull based REST API is more appealing. In REST [2], data and functionality are considered resources, identified by Uniform Resource Identifiers (URIs). REST typically uses a client server architecture with a stateless protocol, usually HTTP. Web applications built on REST architecture are RESTful web services. RESTful Web services map the four main HTTP methods GET, POST, PUT, DELETE to the CRUD actions of create, retrieve, update and delete. REST can be used easily in the presence of firewall. REST and SOAP based web services are platform and programming language independent. Clients and servers are loosely coupled. REST was developed as a simpler alternative to the complicated and verbose structure introduced by the addition of security and message reliability in SOAP.

2.2.6 Tiny Web Services

It is important in the vision of IoT, to allow adding sensors to existing smart infrastructures. This requires network layer and application layer interoperability. Network layer interoperability can be achieved through IP. At the application layer, application developer needs to understand the type of data, parameter values and control messages expected by the sensor. One approach for interoperability is to force each sensor vendor to use new common specification. Another approach is to use existing web services in a lightweight manner. In the paper [11], sensor node reports its interface using Web Service Description Language (WSDL). Applications that want to use the sensor, sends the sensor the specified message. Advantage of this technique is that application developers only need to know the semantics of the sensor, while the WSDL handles the task of generating method calls in a high level language (easy to use). Eg. Visual Studio or Netbeans IDE could be used to parse the WSDL and generate a Java object that provide device messages as function calls with typed arguments. The Visual Studio or Netbeans IDE take care of actual format and packets to be sent according to the WSDL specifications. This allows incorporating a new device, with messages easily sent as automatically generated method calls. The challenges of low battery life of devices, low power could be overcome by having the device implement low power web services on WSDL standard compliant devices. Requirement of sleep for devices to save power could be achieved by using Web Services Eventing [12]. Similar to Tiny Web Services, even in JADE application developers only need to know the semantic of the sensors or the interface for the ‘things’ that they will interact with, while the underneath architecture takes care of generating method call using functions like `call_user_def()`.

2.3 Ideas Supported by JADE

2.3.1 Ambient Intelligence

Ambient Intelligence (AmI) [13] brings intelligence to our everyday environment. It builds on advances in sensors, pervasive computing and artificial intelligence. As a result of growth in these contributing fields, there seems to be a great potential being observed in Ambient Intelligence. Ambient Intelligence needs to be sensitive, responsive and adaptive. An AmI agent perceives the state of the environment through sensors, makes decisions and changes the state of the environment through actions such as robotic assistance or controlling the devices. Sensors are the key to link computational power to physical application. Motion sensors are used to track individuals and motions, however they can not determine who created the movement. An alternative is for persons or items to wear sensors for this purpose. RFID tags are an example of this technology. A combination of RFID readers and motions sensors could produce more precise sensing technology. I-Button are small devices that allow the receptor to communicate with a computer, they are also used as sensors. Microphones and video cameras are used for tracking too. Analyzing the sensor data can be done in a centralized or a distributed model. In centralized model, data is transmitted to central server which fuses and analyzes data, while distributed model provides the sensors with processing capability. Reasoning is used for providing algorithms to provide links between sensing and acting. Reasoning includes modelling user behavior, activity prediction, decision making and spatial-temporal reasoning. To further societal acceptance of

AmI, there is a need to define human centric interfaces that are context aware and natural. Context awareness enables devices that infer current activity of the user and characteristics of the environment, to intelligently manage information content and means of information distribution. [14] proposes an adaptable and extensible context ontology for creating context aware computing infrastructure. It is especially relevant considering the fast evolution in the hardware and software industry, so the decisions made today regarding context specifications should be adaptable and extensible. The ontology provided is a basic, generic context ontology which was built around four main entities. These entities, which include user, environment, platform and service; are mentioned to be based around the most important aspects in context information. The advancements in the field of AmI will go unnoticed if it is difficult for the users. So design of natural interfaces is of greater relevance. AmI could have a significant impact in our lives through many applications in Smart homes, health monitoring, hospitals, transportation, education, etc. Through ambient intelligence a confluence of topics can converge to help society. JADE tries to capture the different aspects of AmI. Analysis of the sensor data is done in a centralized manner in a cloud server. The user could use the browser as a convenient and easy way to interact with the physical system.

2.3.2 Cyber Physical Systems

Cyber-physical systems (CPS) [15] bridge the cyber-world of computing with the physical world. CPS represents a confluence of technologies in embedded systems, distributed systems, dependable systems, real-time systems with advanced micro-controllers, actuators and sensors. CPS must operate dependably, safely, securely and in real-time to aid in the realization of a future societal-scale system. Thereby allowing effective integration and pervasiveness of real-time processing and sensing across heterogeneous logical and physical domains. The availability of low-cost sensors with increased capabilities, high capacity computing devices, wireless communication, ample internet bandwidth and increased energy capacity are the driving forces pushing the advancement of CPS. CPS attempts to interface the powerful and precise logic of computing with the continuous dynamics and noise in the physical environment. CPS are designed to include a network of interacting elements rather than standalone devices. The advancement of Cyber Physical Systems could be vital in realizing the effectiveness of the 'live' variables introduced in JADE. By using the advanced sensors and actuators which are part of the Cyber-Physical Systems, the 'things' could monitor and communicate any change in their state by using the 'live' variable concept developed in JADE, which will be explained in chapter 4. As JADE allows developers to write code in JavaScript for the web and in C/C++ for the 'thing', it could help in bridging the gap between the computational systems and the physical systems

2.3.3 Semantic Technology

In addition to 'thing' and Internet directed vision, the paper presents a semantic oriented vision [16]. Semantic technologies are a means of creating machine interpretable representation which provides an efficient way of sharing and integrating information. Semantic Technology consists of algorithms and solutions that bring structure and meaning to information. Semantic technology helps in understanding and interpreting information. Some examples of Semantic Technologies include Natural Language

Processing (NLP), Artificial Intelligence and Data Mining. The paper suggests using middleware as a software layer abstracting the details of different technologies. Similarly SpaceOS explained in chapter 3, using the cloud as a middleware between the Things and the browser helps in abstracting technology details. Applications need to discover sensors. Semantic Technology is viewed as a key topic to resolve interoperability. Most semantic tools/techniques are created mainly for web resources and often overlook the dynamicity and constraints of the physical world. According to [16], lightweight ontology seem to have a better chance of wide scale adoption. Application of Linked data principles to semantic processing could further improve interoperability. SpaceOS, through the use of JADE, suggests associating attributes to ‘things’, for example the attribute ‘brightness’ could be associated with a lamp and the attribute ‘temperature’ to a thermostat. If we create a link between these two attributes of the lamp and the thermostat, then as the ‘brightness’ of the lamp changes, so will the ‘temperature’ of the thermostat.

Chapter 3

Unified Programming Framework Design

3.1 Design Objectives

A new programming framework dealing with the Internet of Things has several requirements. The framework we propose as part of this research is quite novel and includes elements of all of the points mentioned previously in Chapters 1 and 2. Following are some of the key objectives we wanted to achieve in designing the framework.

1. *Leverage existing trends:* IoT is in a highly dynamic technology sector. Therefore, it is necessary to leverage existing trends in developing the programming framework. In particular, the programming framework should reuse technologies that are heavily used by the developers in the sector.

2. *Lightweight framework:* The framework should have small resource footprints so that it could be ported to highly resource-constrained devices.

3. *Flexible function partitioning:* A programmer of heterogeneous devices such as IoT can immensely benefit from a flexible function partitioning mechanism that allows her to easily deploy custom code at the web or cloud. JADE could be used for calling functions using the `jadef {web}` and the `jadef {cloud}` keyword on the web and the cloud, which run on the environment of the web or the cloud, therefore follow the web CPU cycle and the cloud CPU cycle respectively. While in the case of other protocols like REST and RPC, the function invocation takes place in the system where the REST or RPC call is made. So, the CPU cycle being followed is that of the local system. Using such a functionality provided by JADE, the programmer can easily create computing activities that incorporate the thing, web, and cloud.

4. *Gentle learning curve:* We can expect many programmers to be highly proficient on well known languages. By creating a framework that combine snippets of existing programming languages, can provide a gentle learning curve to the IoT programmer.

5. *Support for heterogeneous systems:* We can expect the IoT ecosystem to contain a highly heterogeneous collection of devices. These devices should interoperate among themselves and cloud and web resident services.

3.2 System Architecture

The unified programming framework developed here relies on a system architecture (also referred to as SpaceOS) that is described below. In this system architecture, we have three major elements, the ‘thing’, the web and the cloud. The functions provided by the architecture are arranged in a stack shown in Fig. 3.1. In this figure, there are three parts, the Core, the Foundation and the Application. The *Core* part has services that run on all elements: thing, web and cloud. These services are responsible for naming, discovery and handling events. When a smart device enters a space, it is given a spaceID by the space resolver. After this it sends a request to the cloud for providing it a name or id (TId) which is unique. The Core also provides Discovery service, which allows a 'thing' to find other 'things' of the same type. Events could be triggered by the 'thing', the cloud as well as the web. For instance an update in the temperature value sent by a thermostat when the temperature recorded by the sensor connected to it changes. This is an example of an event triggered by the 'thing'. Events could be triggered by the cloud using the jacall() function from the cloud on a 'thing'. Users could invoke events on the browser. So, methods to handle the events should be present in the 'thing', the web and the cloud.

The *Foundation* part has services that are specific for the different elements. The foundation services on the thing are responsible for functions such as locationing. Multiple bluetooth markers along with iBeacons could be used for locationing of the 'thing'. The foundation services run on the web and cloud as well. Their functions change depending on the element. The *Application* part has services that are created by the developer using the JADE framework. Each type of component ('thing', web and cloud) has its own version of the stack.

The architecture presented here is cloud-centric, where the nodes in the network communicate through services hosted in a cloud platform. Therefore, it is important to have fault-tolerant communication services built into the 'things' in SpaceOS such that the smart computing environment can at least have partial capabilities even

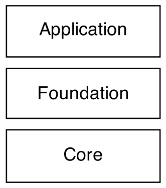


Fig. 3.1 Layered organization of the System Architecture

when the connectivity to the cloud is lost. This could be done in a manner similar to the fault-tolerant model for ‘things’, by having each ‘thing’ store a local copy of all the important information that it requires from the cloud, as previously described in section 3.4.1.

3.2.1 Space OS Interface Implementation

The SpaceOS lies on the application layer, abstracting physical objects connected to the network and creating a logical representation of them. All communication with ‘things’ and web users is done through sockets. For web users we are currently using web sockets and for ‘things’ we use TCP sockets.

Fig. 3.2 shows a sequence diagram of a simple user interaction through the browser. We assume that the smart things are already rendered on the browser. For instance a smart bulb is rendered using an icon of appropriate shape. First, the user selects the ‘thing’ that he/she wants to interact with. Users could select an object in a space by touching the object on the screen on his/her tablet/smart phone or could click the screen on his/her computer screen. This event of touch/click by the user invokes the `GetMenu(x,y)` function, where the parameters correspond to the x and y pixel coordinates of the point where the touch/click occurred. The `GetMenu(x,y)` function fetches the menu from the SpaceOS Server via the Web Server, for the object present

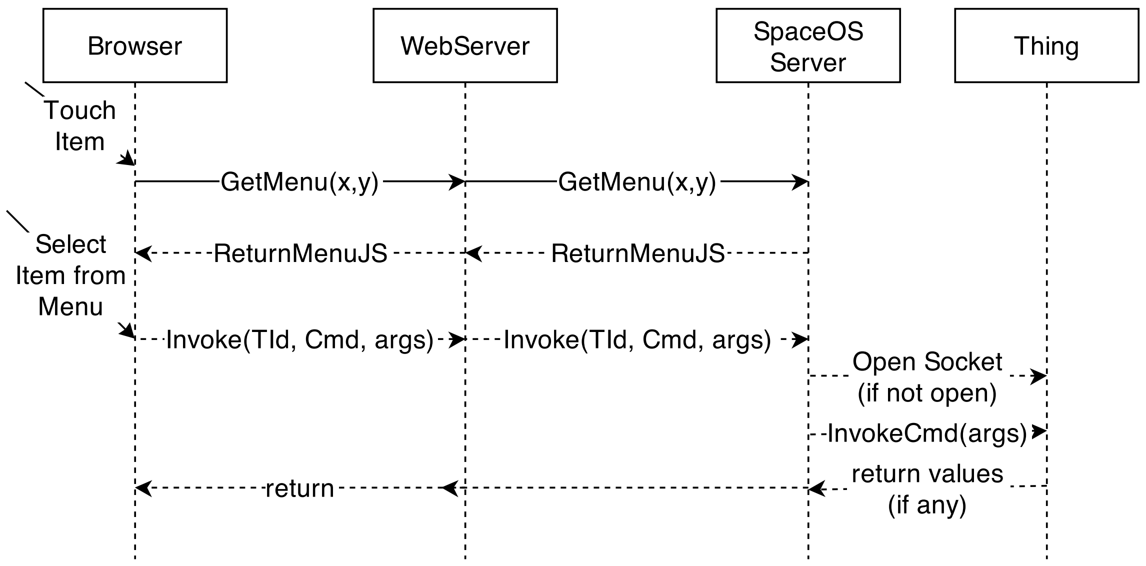


Fig. 3.2 User interacting with the Thing

in that location. The ReturnMenuJS function returns the menu to the browser, which could be viewed by the user. The user then selects an action from the options present in the menu. Once the selection is done, the `Invoke(TId, Cmd, args)` function is called and the action is invoked on the 'thing' via the WebServer and the SpaceOS Server. TId is the id of the 'thing', Cmd is the C function that is to be invoke on the 'thing' and args is the list of arguments for the C function invoked on the 'thing'. An acknowledgement could be sent from the 'thing' to the browser on the successful completion of the action.

3.3 Major Components of the Architecture

Fig. 3.3 shows a break down of the 3 main components of the SpaceOS framework, Spaces (Smart Environments), Web Application and the Cloud Server.

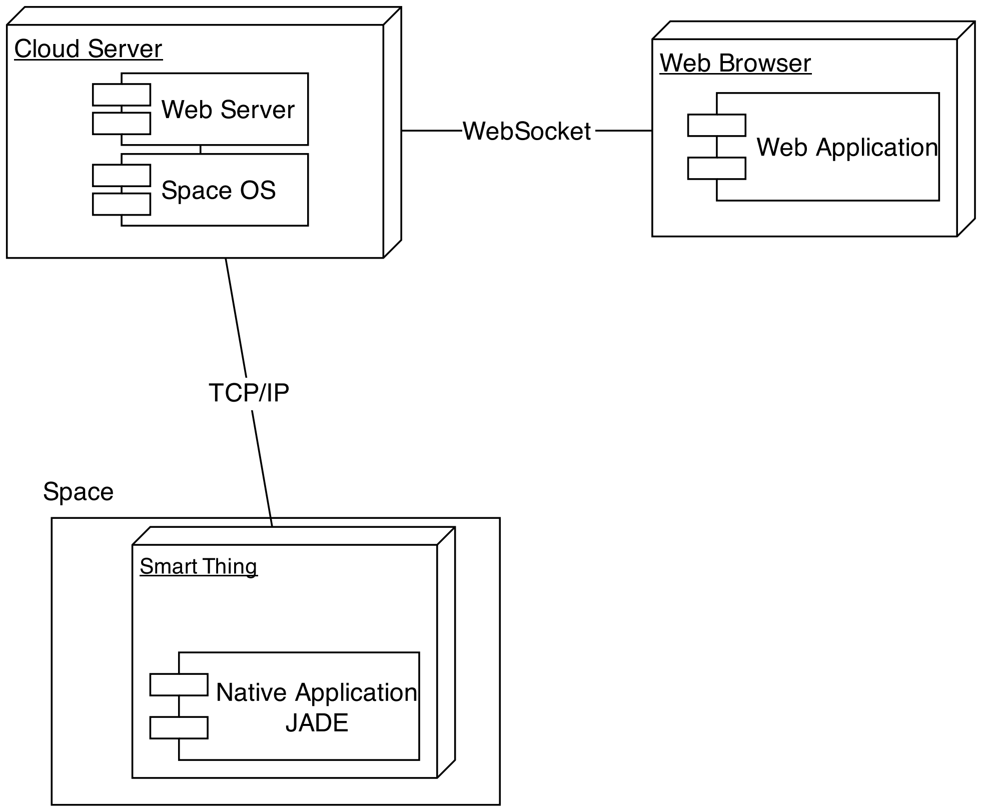


Fig. 3.3 SpaceOS components view

3.3.1 Spaces

A Space is represented by the 'things' that it has and is identified by a unique id. The SpaceOS framework was designed to facilitate the process of adding objects to a smart environment. The act of bringing a smart object into a Space will trigger a chain of events responsible to registering that object with its current space and deploying necessary code to the SpaceOS server, see Fig. 3.5.

Each space has a Space Resolver. Space Resolvers assist the registration of a new object to a specific space in the network. For example, lets assume we want to make our house a smart environment. By bringing a smart object into the house it will connect to the local network. The smart objects will have the framework code installed which will allow it to do a series of discover protocols. The smart object will send a broadcast message asking if there are any resolvers in the network. If yes, the resolver will reply by sending the unique id of their space. By receiving this unique id the new smart object can register itself with the SpaceOS server. In the case where a resolver is not found, the new smart object will register itself with the SpaceOS server and then becomes the resolver for the space. The subsequent paragraph and figures 3.4 and 3.5 describe the 'thing' registration process in more detail. Once the registration is done the smart object and the SpaceOS server start the synchronization phase. In the synchronization phase the smart object will subscribe for events (see Section 3.3.3.2) and publishes its metadata, live variables and server side code (see Chapter 4).

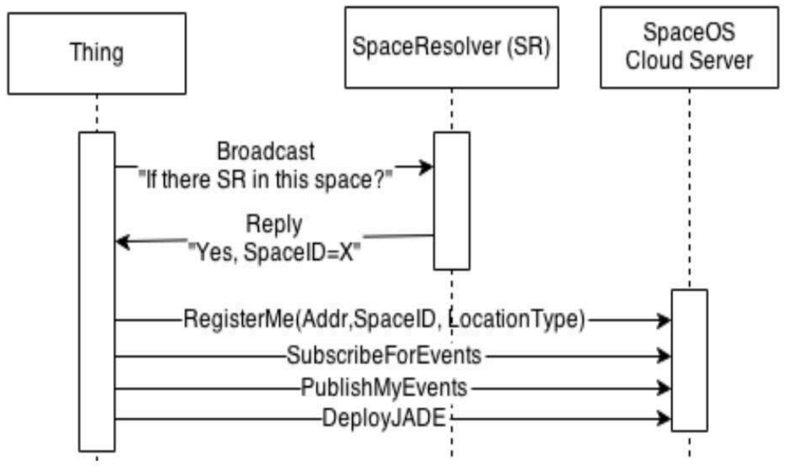


Fig. 3.4 Thing registration phase

Thing registration process When a 'thing' enters a space, it sends a broadcast message to all devices in the network asking whether there is a space resolver in that space. If a space resolver is already present, the space resolver replies with a message 'yes' to the new device followed by the space id of the space. Figure 3.4 shows the exchange of messages when a space resolver is present in the space. The 'thing' then requests to be registered in the space by sending a RegisterMe(Addr, SpaceID, LocationType) message to the Space Resolver using the Space ID provided by the Space Resolver. Once this is done, the 'thing' could Subscribe for events, publish events and deploy JADE.

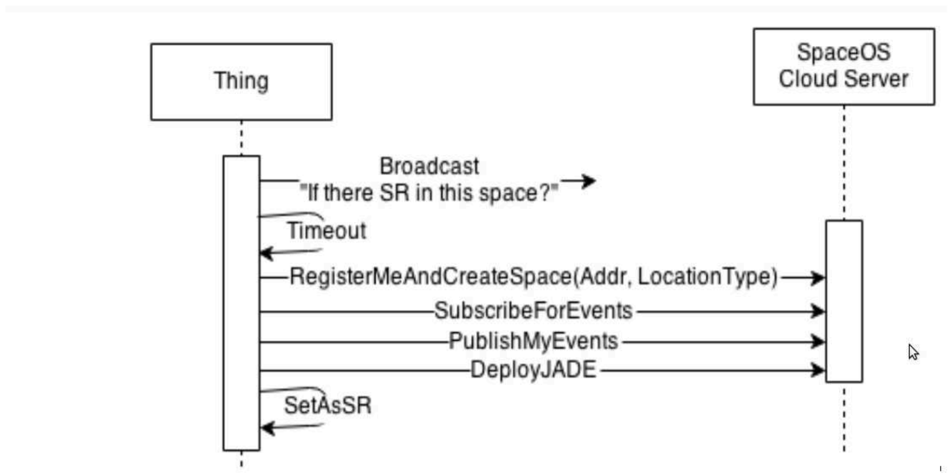

Thing
SpaceOS Cloud Server
Broadcast "If there SR in this space?"
Timeout
RegisterMeAndCreateSpace(Addr, LocationType)
SubscribeForEvents
PublishMyEvents
DeployJADE
SetAsSR

Fig. 3.5 Thing becomes Space Resolver

When the broadcast message asking for Space Resolver in the space is timed out, the 'thing' realizes that the space does not have a Space Resolver. This scenario is shown in Figure 3.5. Once the timeout occurs, the 'thing' needs to create a space and then register itself. Next, the ‘thing’ could subscribe for events, publish events and deploy JADE. The ‘thing’ then sets itself as the Space Resolver for that space.

3.3.2 Web Application

The user browser is the interface at which the user will interact with the smart environment. It has two modes, the interaction mode and the programming mode. In the interaction mode the user is able to visualize smart objects in the environment, raise events from objects and interact with other users connected. In the programming mode users can create relationship between objects and events. These relationships are used when smart objects notify the event manager that an event has happened. The act of building relationships between physical smart objects is what we call ”programming the physical world.

3.3.3 Cloud Server

The SpaceOS server is responsible for managing spaces and coordinating interactions between user-to-thing, thing-to-thing. It has 3 main modules which are explained below.

3.3.3.1 Space Management

The Space Management module is responsible for managing the logical representation of spaces, objects and users. Spaces, objects and users can be added, modified or deleted as necessary. It is also responsible for keeping track of which spaces and objects are still available (online). This is done by probing the smart objects and verifying that they are alive. If no object is available for a particular Space, that

Space is considered offline, and is removed.

3.3.3.2 Event Management

Each smart object implements the observer pattern. Objects can subscribe for different types of notifications based on object type, content type or for an specific event raised by a specific object. Developers are able to expose their events at the synchronization phase (see Section 3.3.1). As events are raised, the event manager receives them through the network and is responsible for finding and notifying the subscribers with the assistance of the space manager. For example, if in our smart environment we have a switch that raises two events, SwitchUp() and SwitchDown(). A smart lamp can register for such events and at the same time it could specify a callback function, such as TurnOn() or TurnOff(). Therefore, relationships between events can be built between smart objects of the space.

3.3.3.3 View Management

The view management is responsible for handling all request and responses sent to the user browser. It communicates with the browser through web sockets. When requested, It will communicate with the space management to gather information about the smart objects available and their functionality. It's also responsible for forwarding event requests made by the user. For example, the user can request the lamp to TurnOn() through the web interface. The view management will receive that and forward that request to the Event manager for processing. Thus, all actions regarding user interface and user interactions are handled by the view manager.

3.4 Important Issues in Realizing the Architecture

We need to investigate many issues related to reliability and security before realizing this architecure. Although the different ways of ensuring reliability and security have not been implemented yet, in this section we propose different methods to deal with these issues. Reliability is an important issue while considering the design of a unified programming framework. As JADE is inspired from the RPC model, we first compare the reliability offered by JADE to that provided by RPC. Next, we look at different techniques for improving the security.

3.4.1 Reliability

An application using RPC may be needed to be aware of the kind of transport protocol that is running underneath. For instance, in the case of a reliable transport protocols like TCP, most of the reliability issues are already taken care of by TCP. However, if the application is running on top of an unreliable transport protocol like UDP, the application must implement its own timeout, retransmission and duplicate detection policy.

There are three different forms of RPC call semantics, at least once, at most once and exactly once. In the case for at least once, client stub re-transmits request until a valid response arrives. At least once semantics are common for Idempotent procedures. Idempotent procedures are those that have no additional effect if called more than once. Consider an application that is using unreliable transport such as UDP, if the application retransmits messages after timeouts and does not receive a reply, it can not infer anything about the number of times the procedure got executed. However, if it receives a response, it could infer that the procedure got executed at least once. Exactly once semantics for RPC is similar to local Inter Process Communication (IPC), however it is harder to achieve because of server crashes and network failures. If an application is using reliable transport such as TCP, on receiving the reply message the application can infer that the procedure got executed exactly once. However, it can not infer anything if it does not receive a reply. For the case of at most once, it is important that the server remembers the previously granted requests from a client and not regrant them. To achieve this, a server may use the Transaction id that is packaged as part of the RPC message. A client application may want to reuse a transaction id when transmitting a call. The server may wish to remember the id of the executed call and chose not to execute further calls with the same id in order to ensure at most once semantics.

In contrast to RPC, 'live' variables which are part of JADE are totally "best effort". There is no notion of reliability. There could be network failure, server failure and 'thing' failure. So 'live' variables are UDP and do not maintain any state at the receiver or sender. Duplicate, lost ,reordered and delayed messages are possible while using live variables.

Currently, for JADE, we are using TCP as the communication protocol, thereby ensuring that there are no network related failures. However, it is important to safeguard against failures that could occur at the server and the 'thing'. Each 'thing' could store a local copy of all the important information relevant to it, that is stored in the server, like names or the id (TId) of other 'things' in the network, the events that the other 'things' have subscribed for, so that even during the event of server failure, there is partial functionality in the 'thing', to ensure that the whole setup does not crash. There are several ways of implementing fault tolerance for the 'things'. One of the ‘things’ in the network could be made the ‘central node’, which sends periodic broadcast messages to all other ‘things’, to check whether they are alive. Failure of a ‘thing’ is detected if a timeout occurs at the central node. Another method could be to use a peer to peer mechanism for tracking the failure of a ‘thing’ using a Distributed Hash Table. Each ‘thing’ checks the liveness for one other ‘thing’ in the network and stores the “Tid” which is the id of the ‘thing’ being being monitored, in its local hash table; this ensures all the ‘things’ in the network are being monitored. Initially, when there is only one ‘thing’ in the network it is the only one being monitored. When the second ‘thing’ enters the network, the first ‘thing’ monitors the second and the second monitors the first. When one more ‘thing’ enters the network, it is monitored by the first ‘thing’, while it monitors the second ‘thing’. The first ‘thing’ sends a welcome message containing the Tid of the second ‘thing’ to the newly entered ‘thing’, which stores this Tid in its local hash table and sends a reply message to the first ‘thing’ with its own Tid. So, that is how the first ‘thing’ will now monitor the new ‘thing’, the new ‘thing’ monitors the second ‘thing’ and the second ‘thing’ is left unchanged and continues to monitor the first ‘thing’. The chain continues, whenever a new ‘thing’ enters the network, it is monitored by the first ‘thing’ while it monitors the ‘thing’ that was being monitored by the first ‘thing’ before. The second ‘thing’ is at the end of this chain and it monitors the first ‘thing’.

3.4.2 Security

In the architecture presented here, interactions take place between the three main components, the ‘thing’, the web and the cloud. Intrusions and interference could take place at any of these components. Therefore an authentication needs to be done between the different components involved in the interaction to ensure that there is no breach in the security. For instance, if a ‘thing’ connects to the web, there should be an authentication done by the web server to ensure it’s the correct ‘thing’ that it is connecting to. This could be done by checking the ‘thing’ id from the list of ids in its directory to determine whether the ‘thing’ is a part of the existing network. Similarly, when there is interaction involving the ‘thing’ and the cloud, the cloud server should authenticate whether the ‘thing’ that is interacting with it belongs to the network. Before a user accesses the web, and tries to access the cloud through the web, an authorization step should take place to determine the permissions and the level of access that should be granted to the user. In addition there should be a logon authentication of the user to ensure that the user is authenticated before entering the network. For accessing the web, password authentication or biometric authentication could be done for the user, based on the kind of device (smart phone, tablet or computer) from which the user tries to access.

Chapter 4

JADE Programming Framework

4.1 Overview

JADE is a programming framework to facilitate the interaction between the ‘thing’, the web and the cloud. Figure 4.1 shows how the interaction between these different components takes place through the different functions provided by JADE. `jadef {web}` is used for interaction between the ‘thing’ and the web. As JavaScript is the main language of the web and also the language for the cloud server, `jadef {web}` and `jadef {cloud}` allow developers to write code in JavaScript syntax, which is later inserted into the web or the cloud. The cloud server is written in NodeJS, the developer could insert JavaScript functions for the cloud by using `jadef {cloud}`. JADE allows a publisher-subscriber system; ‘thing’ exports to the cloud the events it subscribes for, by using the `jasubscribe` function and publishes events by using the `jaevent` function. So, the cloud could keep track of the subscribers and publishers for different events. `jacall` function exposes the functions that could be called on the ‘thing’ from the cloud.

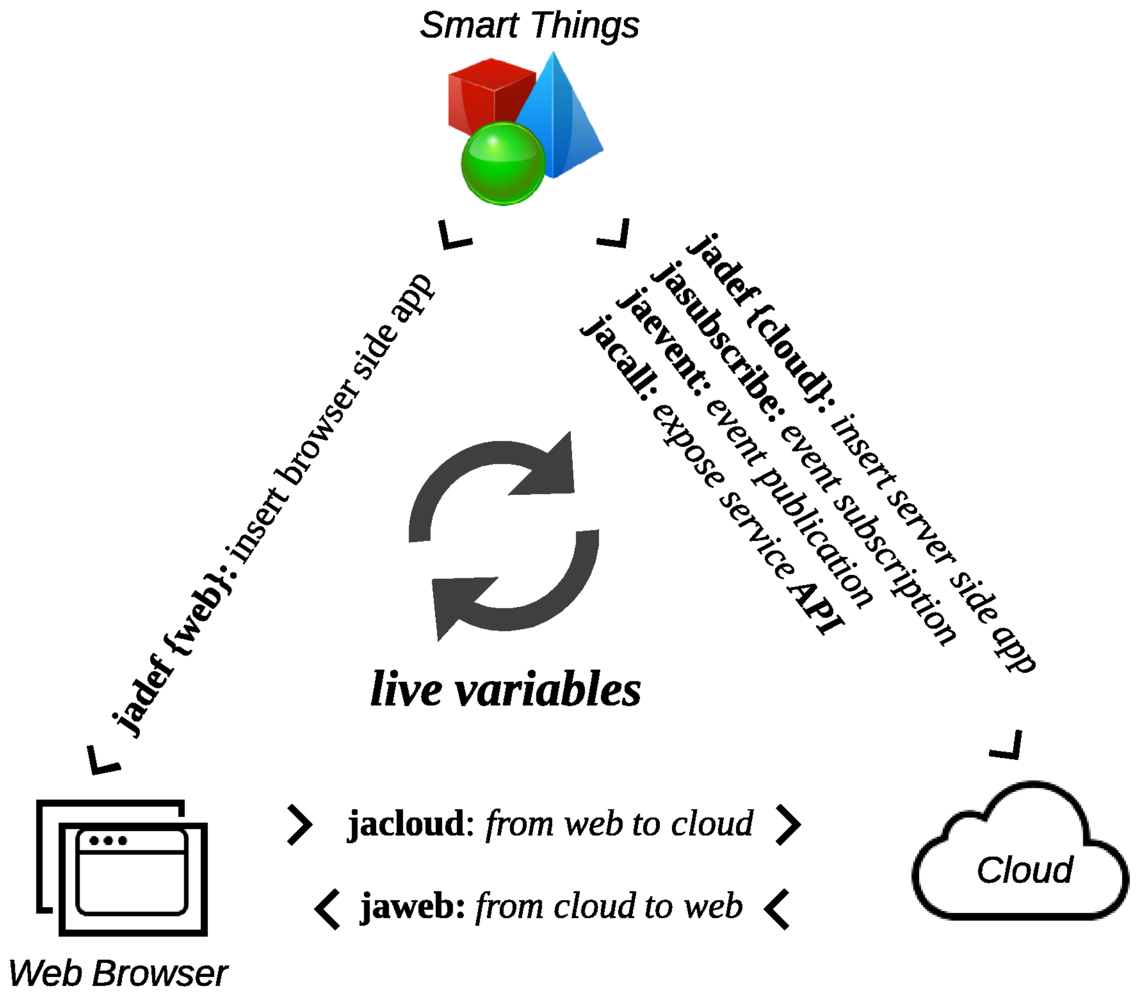

Smart Things
jadef {web}: insert browser side app
jadef {cloud}: insert server side app
jasubscribe: event subscription
jaevent: event publication
jacall: expose service API
live variables
Web Browser
jacloud: from web to cloud
jaweb: from cloud to web
Cloud

Fig. 4.1 'thing', web and cloud interaction through JADE

The interaction between the web and the cloud is enabled by using the `jaweb` function for calling the web from the cloud. `jacloud` is used for calling the cloud from the web. The developer could introduce the `jaweb` function invocation inside the `jadef {cloud}` function, that goes to the cloud server and the `jacloud` function invocation inside the `jadef {web}` function, which goes to the web.

A 'thing' could have have values which change dynamically or could be changed by the user, these variable are 'live' variables. 'live' variables could be invoked at the ‘thing’ if the value associated with a ‘thing’ changes, this change in the value needs to be passed to the cloud and the web. While the user could change the ‘live’ variable value through the web, again the change needs to be reflected on the real ‘thing’ through the cloud.

4.2 Functions

4.2.1 jadef C Function

JADE language allows the developer to code for both the server and the web. The code that is to be passed to the server or the browser, needs to be enclosed within a function. Such a function should begin with the keyword ‘jadef’, followed by either the keyword ‘web’ or ‘cloud’ enclosed within parentheses depending on where the function will be executed. The function name and parameters follow the ANSI C syntax. Inside the function’s body, the code should be written in standard JavaScript syntax. This function is called from the ‘thing’ and runs in the cloud or the web. Below is the syntax for writing `jadef` function for the web:

```
jadef {web} C_function_prototype()
{
    // JavaScript body
}
```

Listing 4.1 jadef function syntax

A simple example `jadef` function obeying the syntax is provided. Notice the C function declaration and JavaScript function body.

```
jadef {web} foo(int xValue)
{
    var x = liveVar(‘‘foobar.x”, xValue);
    x.onupdate = function(){
        alert(‘‘new value of x ” + x.value +
            ‘‘baz: ” + baz);
    };
}
```

Listing 4.2 jadef function example

The cloud server for JADE will be implemented in NodeJS, which runs the JavaScript code from the ‘thing’. The ‘thing’ should also provide its name, so that the functions in the cloud can be associated with the corresponding ‘thing’. For sending the JavaScript code to the cloud, the developer should use an additional keyword ‘cloud’ enclosed within parentheses in front of the function name, as shown below:

```
jadef {cloud} C_function_prototype()
{
    // JavaScript body
}
```

Listing 4.3 jadef server synatax

JADE provides the feature of creating multiple virtual instances of the same physical object. The syntax for achieving this functionality is shown below:

```
jadef {thing} {//choice}
{
    // C body
}
```

Listing 4.4 jadef ‘thing’ syntax

The choice could be exclusive, mutual, weighted or time slice. This is further elaborated in Section 4.5.

4.2.2 jarequire

The JavaScript body in the `jadef` function may require some external libraries, global variables or function definitions which could be included inside jarequire(). If the user wants to insert HTML5 templates or link CSS files into the HTML5 templates, jarequire could be used for this purpose too.

```
jarequire()
{
    // external libraries
    // global variables
    // global functions
    // HTML5 templates.
    ...
}
```

Listing 4.5 jarequire syntax

In example below, the variable ‘baz’ is defined in the global scope, which allows the `jadef` function `foo` in Listing 4.2 to access the value.

```
jarequire()
{
    var baz = 10;
}
```

Listing 4.6 jarequire example

4.2.2.1 jasubscribe Function

Allows the developer to write a function in the C/C++ syntax in the 'thing' that gets executed when the event is triggered by another 'thing', browser, or the cloud server. Below is an example of the prototype of the jasubscribe function:

```
jasubscribe {event name} c_function()
{
    // code to be executed when the event occurs
}
```

Listing 4.7 jasubscribe function syntax

4.2.3 jacall Function

Using the 'jacall', the developer can mark a function as callable from the cloud or another 'thing'. A JavaScript function signature that corresponds to the C function will be registered at the cloud server as an API supported by the 'thing'. This API can be discovered by other cloud resident functions and invoked, which will result in a callback to the thing.

```
jacall C_function( )
```

```
{
    // jacall function calls to interact with
    // other 'thing's through their exposed API
    // 'thing' application code ...
}
```

Listing 4.8 jacall function syntax

4.2.4 jaevent Function

As we are using a publisher-subscriber model for event handling, there needs to be a mechanism to publish events to the server so that 'jasubscribe' will receive the events. The 'jaevent' function allows the developer to define new events or use predefined events in the system. Below is the syntax for jaevent:

```
jaevent {event_name} {
    // declare variables associated with the event
}

void c_function(){
    // assign values to the variables associated
    // with the event
    raise_jaevent(event_name,
        variables_as_parameter);
}
```

Listing 4.9 jaevent and raise_jaevent function syntax

The call to `raise_jaevent` transfers the event to the server. The server notifies the subscribers of the event when it gets published. At the system level, a data structure is maintained for storing the events and its parameters. A hierarchical naming convention is followed underneath to help identify different events. For instance "event.user.eventButton" could be used to identify an event called 'eventButton' which is created by the developer. While "event.system.mousemoveEvent" could identify a system event that is used by the developer.

4.2.5 jaweb and jacloud Functions

The programmer may want to write code segments which when run on the 'thing', is transferred to the web and could invoke functions on the cloud from the environment of the web. Similarly parts of code which when run on the 'thing', could be transferred to the cloud from where it could invoke functions on the web. To allow the programmer to achieve these desired objective we provide two functions `jacloud` and `jaweb`. `jacloud` and `jaweb` will be used for calling the cloud from the web and the web from the cloud respectively. Hence, the invocation to the `jacloud` and `jaweb` functions could be included as part of jadefweb and jadefcloud respectively.

So, the JADE parser while parsing the JavaScript body inside `jadef {web}` could search for an additional keyword 'jacloud'. Once it encounters 'jacloud', the parser will replace jacloud with ws.send(Arguments) or equivalent NodeJS command for websocket communication between the cloud and the web.

Below is the syntax for calling the `jacloud` function from the `jadef {web}` function.

```
jadef {web} C_function( args )
```

```
{
    // JS body
   jacloud cloud_function( arg )
}
```

Listing 4.10 jacloud syntax

After being parsed by jade.py, which is the JADE parser, the output will look like what is shown below.

```
C_function(args)
{
   // JS body
   ws.send("cloud_function", arg);
}
```

Listing 4.11 parsed jacloud function

Similarly, we do the invocation for `jaweb` function inside the `jadef {cloud}` function.

```
jadef {cloud} C_function( args )
{
    // JS body
   jaweb web_function( arg )
}
```

Listing 4.12 jaweb syntax

Below is the ouptut after being parsed by the JADE parser.

```
C_function(args)
{
    // JS body
    ws.send("web_function", arg);
}
```

Listing 4.13 parsed jaweb function

4.3 Live Variable

JADE allows the programmer to *tie* a variable in the 'thing' to a variable in the cloud or web. We believe this provides a much simpler way of obtaining the latest value of a variable without generating another event for it. To achieve this, we introduce the concept of "live variables."

4.3.1 Update triggered by the 'thing'

Variables whose value could change are referred to as 'live' variables. They are denoted by the storage class `live` we introduce into the C side.

```
void foobar(){
     live int x;
     x = 5;
     foo(x);
}
```

Listing 4.14 An example of live variable

The ‘live’ here is similar to the static or extern keyword in C. When a variable is declared ‘live’, if the value of the variable changes elsewhere, the new value will be reflected locally, and vice-versa. While writing the code in JADE, any portion of the code written in C syntax, could include a ‘live’ variable, except the JavaScript part which is inside the jarequire and the jadef function for the cloud and the web.

Inside the ‘jadef’ function for the cloud/web, the developer could add the event that should transpire in the cloud/web, when a live variable value update is received from the ‘thing’, using the syntax shown below.

```
x.onupdate = function() {
    // event upon value update
};
```

Listing 4.15 onupdate JavaScript function

The onupdate function allows the programmer to execute arbitrary JavaScript code when the value of the live variable gets modified in the ‘thing’.

4.3.2 Update triggered by the web/cloud

The live variable can be updated in either end. Above, we described how the updates propagate from the ‘thing’ to the cloud or web. Below we describe how updates propagate from cloud or web to the ‘thing’. In the C side, JADE provides a library function called `checkVal()` that allows the programmer to read the current value of the live variable. If there is no updates for the live variable, the `checkVal()` routine will timeout after the given interval. The C function uses the return value to indicate the presence of a updated value.

```
void updateThing()
```

```
{
    live int x;
    while(1) {
        int timeout = 1000; //1000 ms
        int xflag = checkVal("x", timeout);
        if(xflag) {
            // new value found for x
        } else {
            // timeout occurred
        }
    }
}
```

Listing 4.16 update ‘thing’s’ live variable

4.4 Implementing JADE

The JADE implementation has two major parts. First is the JADE preprocessor that converts JADE source file to the C/C++, ino (Arduino compatible C/C++), and JavaScript. Second is the JADE runtime that is necessary to integrate all the components together. This part is mainly responsible for message communications and invoking the necessary functions.

4.4.0.1 The JADE Preprocessor

The JADE Preprocessor is written in python. It takes a JADE file with .ja extension as input and generates a .ino or .c and JavaScript files as output. The preprocessor parses the JADE file one line at a time and adds the code to the JavaScript file or the .ino (or .c) file determined by the presence of keywords like `jadef`, `jasubscribe`, `jacall`, `jaevent` and `jarequire`.

4.4.1 The JADE Runtime

The JADE runtime is made of a library in C, library in JavaScript, and a NodeJS server (currently server is implemented in Python). The C library consists of important functions that could be used by the developer while writing the C/C++ part of the code in the JADE language. The C and JavaScript libraries are responsible for implementing the remote procedure call (RPC) functionality so that C functions can call their JavaScript implementations and vice-versa. In addition to the RPC functionality, the libraries also support the live variable updates through a similar but simpler mechanism. In live variable updates, we need to keep track of the functions getting in scope and leaving the scope. Further, the updates for the live variables need to be routed to the appropriate variables when many variables are active at the same time.

One of the features of the JADE runtime is its ability to work with web browsers. That is, the Javascript programs running inside web browsers can use the JADE runtime to communicate with programs running in 'things' and cloud.

4.4.2 hlib

It is a library that is written entirely in C. The hlib is present on the ‘thing’. The library consists of important functions that could be used by the developer while writing the C/.ino part of the code in the jade language. Originally, hlib was meant to be used for the W12 Window system, containing a set of drawing primitives, to be used by application developers, to render on the screen when the appropriate event occurs. The hlib assists developer by allowing them to focus on the application and not worry about the communication channel and message transfer to the server. The hlib could be included as part of the header in the jade program, by following the C syntax for including a file. The hlib directory consists some important files for events and commands. hlib stores a the set of events that could occur at the browser / server, based on which the event received from the server are identified and appropriate handler is called. In event.h, the header file for events, we enumerate all the events that could happen like ClickEventType, MouseDownEventType, MouseMoveEventType, MouseDragEventType et cetera are some examples of events. Event is registered as a structure containing the event type and the event value which is a union, comprising of keyboard, mouse values et cetera. Prior to creating the `jsevent` function in JADE, event handling was being done by registering callbacks for events.

4.4.2.1 call user def Function

The application developer may want to create functions in javascript for the web or the cloud, or may want to call some pre-existing javascript function. The javascript function is invoked by using the `call_user_def` function call in the C/.ino program. The `call_user_def` function present in `hlib.c`, is a special purpose function, which takes the function name and function arguments; and calls the corresponding function at the web or the cloud. For the ease of programming, the developer does not need to bother about invoking `call_user_def`, rather it is inserted by the preprocessor. The biggest challenge in creating the `call_user_def` function was to allow variable type and number of arguments to be passed. To facilitate this functionality, we used `va_list`, and for interpreting the type of the parameter, an additional single character parameter needs to be passed to the function. The character 'i' identifies integer, 's' for string, 'c' for character, 'f' for float and 'd' for double. Below is the prototype for invoking the `call_user_def` function:

```
call_user_def(``atest", ``ss", xstr, ystr);
```

Listing 4.17 Example invocation of call user def funciton

The first parameter is the name of the javascript function which we want to be called at the server or the web. Next is the type of the parameters which are needed by the function followed by the parameters themselves. Here, the parameter ss implies that the two arguments 'xstr' and 'ystr' are strings. In the next step, the function name is wrapped in a JSON object and sent to the server.

4.4.2.2 update function

This function is responsible for updating the value of the live variables every time a new value is assigned to them. The function is inserted by the preprocessor into the C/.ino file to update the value of a variable in Javascript. Below is the prototype of an update function:

```
update(''main_y'',''s'',main_y);
```

Listing 4.18 Example invocation of update funciton

The argument that update function receives the variable, the type of the variable and the value of the variable.

4.5 Virtualization of Things

In addition to providing a unified programming model, JADE also introduces a novel concept of virtualization of things such that the mapping between a physical 'thing' and its virtual counterpart is no longer one-to-one. This idea of 'thing' virtualiztion was first mentioned by Saracco in [17]. It should be noted that the virtualization we envision here is little different from the common notion of virtualiztion described in existing literature [18], where a virtual object is considered as the dynamic representation of its physical counterpart.

With IoT, the nature of interaction between human and the real world is no longer limited to the physical one, we could have users visiting a real world through the web. While those visitors remained passive observers earlier, they could be active by trying to control the things through the web. Such activities create a big problem for the application developers due to the concurrent access of many users. The JADE approach is to simplify the problem by offering a virtualization idea.

Existing approaches for connecting physical objects to cyberspace such as AllJoyn [5] do not handle the problem of concurrent access. By enabling virtualiztion, JADE could introduce new possibilities for using physical spaces. For example, this extreme form of virtualization could enable a "service-oriented" access to physical spaces and things.

The virtualization of a 'thing' means that a physical object can be assigned multiple virtual entities such that these represent the virtual instances of the physical object. With multiple users, each user can acquire their own copy of the virtualized 'thing', with full interfacing functionality. As shown in Figure 4.2, the virtualization of the thing can be achieved using a simple hypervisor implmentation on the thing.

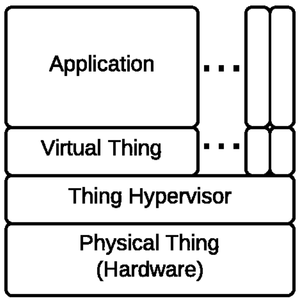


Fig. 4.2 Layered virtualization stack

Each virtual object can be mapped to its physical object under one of the following virtualization models.

In JADE, several virtualization models are offered to the developer. Based on the property of the physical object, and the nature of the application, the developer can select from: exclusive, majority, weighted, and time slice. With the exclusive model, one virtual instance monopolizes over the interface until it exits, thus no shared user control is present. With the majority model, a majority representation over all of the virtual instances operates the physical object, and shared user control is achieved. With the weighted model, which is a general case of the majority model, each virtual instances have their own associated weights, such that the user privilege can be reflected through the weights of individuals' virtual instances. With the time slice model, analogous to process scheduling, virtual instances are allocated a time slice, during which the virtual instances are imposed on the physical object.

In JADE, through the following syntax, the 'thing' can be designed to take on the different virtualization models. In the example in Figure 4.3, the expresso machine should under an *exclusive* model while the lamp could be using any one of the *shared* model.

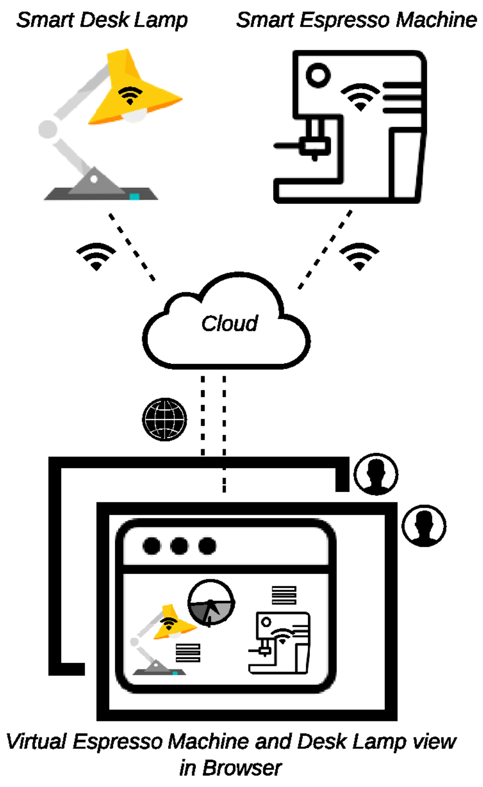


Fig. 4.3 Time slice virtualization model

```
jadef {thing} {virtualization model}
{
    // 'thing' application code
}
```

Listing 4.19 jadef thing virtualization syntax

Chapter 5

Experimental Results

In order to demonstrate the capability of the JADE programming framework, we choose to implement an example prototype application over the Intel Galileo [19].

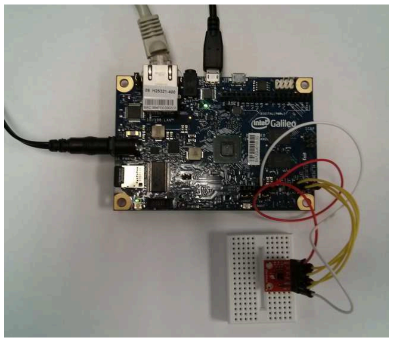


Fig. 5.1 Intel Galileo with accelerometer prototype

The prototype, shown in Figure 5.1, consists of an Intel Galileo connected with an MMA8452Q accelerometer [20]. As an example, a motion control based web game, shown in Figure 5.2, where the user is required to maneuver the bird into the labeled goal zone has been designed. Through the use of live variables, initialized in Listing 5.1, real time accelerometer movements can be computed and automatically updated to the browser, as in Listing 5.3. On the browser side, the live variable `angle_X` is tied to the live variable `tiltAngleX` on the 'thing' side such that desired actions upon update can be performed through JavaScript functions, shown in Listing 5.4. Example codes show only x-axis computation, it is analogous for y and z.

```
live int angleX = 0;
live int angleY = 0;
live int angleZ = 0;
```

Listing 5.1 Live variable declaration

```
void setup() {
    Ethernet.begin(MAC);    // Establish ethernet connection
    // Establish connection to server machine
    move(angleX, angleY, angleZ);
}
```

Listing 5.2 Ethernet and server connection setup

```
void loop()
{
   angleX = computeAngleX();
   angleY = computeAngleY();
   angleZ = computeAngleZ();
}
```

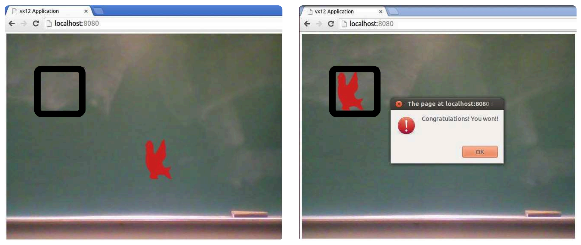
Listing 5.3 Compute title angle in real time

```
jadef move(int Xvalue, int Yvalue, int Zavalue)
```

```
{
    var angle_X = liveVar("loop.angleX", Xvalue);
    var angle_Y = liveVar("loop.angleY", Yvalue);
    var angle_Z = liveVar("loop.angleZ", Yvalue);
    angle_X.onupdate = function(){
    // x direction movement: angle_X.value
    }
    angle_Y.onupdate = function(){
    // y direction movement: angle_Y.value
    }
    angle_Z.onupdate = function(){
    // z direction movement: angle_Z.value
    }
}
```

Listing 5.4 JavaScript function to be executed on the web browser

Including shown code snippets above in the `.ja` file, through the JADE preprocessor, a `.ino` file and `.js` file will then be generated. The developer can compile and upload the application to Arudino, and perform desired actions upon real time update from the accelerometer.



Congratulations! You won!

Fig. 5.2 Game Initial State (Left) and Game Goal State (Right)

5.1 Design Evaluation

To evaluate the design of the system architecture in practice, we measured the memory footprint, network transmission efficiency, and network transmission performance between a cloud server and a thing prototype. For the measurements, we used a PC with UBuntu 14.04 running a python server and tools like network analyzer, and an Intel Galileo [19] with JADE runtime deployed as the thing prototype. The program on the thing has been compiled with i586-poky-linux-uclibc-gcc (GCC) version 4.7.2. With memory footprint, device of interest is the Intel Galileo as other devices are not typically resource constrained. The link between thing and cloud is Ethernet, with all devices reside in LAN. The message exchanging protocol at the transport layer is TCP. In terms of M2M communication, JADE resides between the application layer and transport layer in the Internet stack. For transmission efficiency, we measure the efficiency of using JADE as service for communication between IoT devices. The efficiency is measured for different payloads from 0 to 512 bytes passed by the application layer. For transmission performance, we measure the latency of 10,000 messages for different payloads from 0 to 512 bytes. During the evaluation, payload is pushed to cloud in the form of RPC, with payload being the parameter.

5.1.1 Memory Footprint

Intel Galileo board runs Yocto based Poky Linux distribution, which supports native Linux based applications. *smem* is used to analyze the memory footprint, and **PSS** is measured. The simulation program on the thing uses JADE for event management and RPC. The goal is to examine memory usage under different loads. The simulation on the thing resembles sensor activities in typical IoT settings, where a sensor node collects data and pushes to the cloud, while periodically receives data from the cloud. The sending rate from thing to cloud varies from $1Hz$ to $100Hz$, and from cloud to thing stays constant at $0.2Hz$. The sending rate will be referred as activity, and is normalized against $100Hz$ in column 1 below. The payload size is fixed to 100 bytes both ways.

The memory overhead incurred by making JADE library calls at different rates is examined. In both scenario, simulation program collects data after receiving data from cloud, and simultaneously, pushes data to cloud in the form of RPC at different rates. The *difference* is, in **Scenario A**, the JADE library function used to push data to cloud is *void*; but is *valid* in **Scenario B**. The measurement shows *proportional set size* measured for the simulation process. Under light activity, JADE library calls incur insignificant memory overhead. Whereas, under moderate and heavy activity, overhead is 4 kB.

5.1.2 Network Transmission Efficiency

The simulation program on the thing pushes payload of different sizes to cloud through JADE. The goal is to examine data transfer efficiency using JADE as service for communication. For each message, the size of payload passed by the application layer to JADE is compared against size of payload JADE passed to the transport layer.

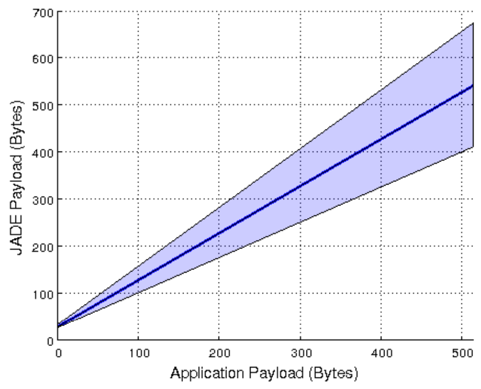


Fig. 5.3 Data Transmission Efficiency

JADE encodes each message to be passed in the JSON format [21], which incurs a slight transmission overhead. The JSON object is transferred over the network as a string and is evaluated at the receiving end.

5.1.3 Network Transmission Performance

The latency is measured in terms of TCP round-trip time for different payloads from thing to cloud.

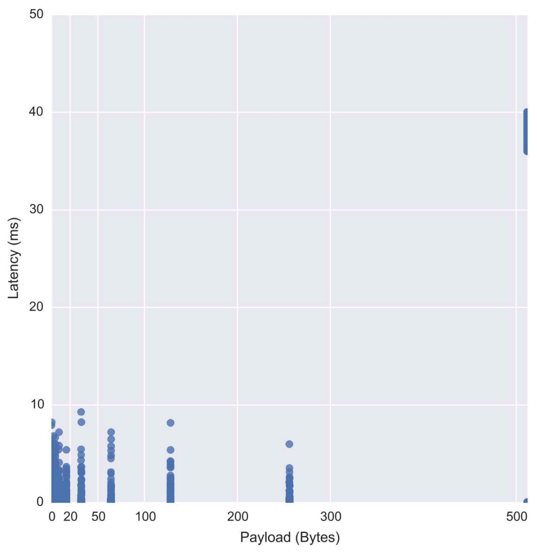


Fig. 5.4 Network Transmission Latency

As the size of the payload increases, the latency also increases. The deviation is a result of standard network congestion and TCP congestion management.

Activity	Scenario A	Scenario B	JADE API Overhead
1%	144.0 kB	144.0 kB	∼0 kB
10%	144.0 kB	144.0 kB	∼0 kB
20%	144.0 kB	144.0 kB	∼0 kB
50%	144.0 kB	148.0 kB	∼4 kB
100%	148.0 kB	152.0 kB	∼4 kB

Table 5.1 Memory usage for simulation program under different loads

Chapter 6

Example Applications

6.1 Internet of Things

We are in a process of a gradual transition from the Internet of Computers to the Internet of Things (IoT) [22]. The reduction in the size, price and energy consumption of processors has resulted in them being widely used and integrated into everyday objects. “Smart” objects play a key role in the IoT vision. Using sensors, the smart objects are able to perceive their context, while built-in networking capabilities allows them to communicate with each other, access internet services and interact with people. More devices like sewing machines, exercise bikes, washing machines, light bulbs and thermostats are being computerized and fitted with network interfaces. Various technical developments taken together help in bridging the gap between the virtual and the physical world. These technical developments include providing effective communication technologies and wireless technologies like GSM, Wi-Fi, Bluetooth, Zigbee and more. Addressability for objects, including discovery, look-up or name services, thus providing remote configuration. Objects could be identified by using RFID, NFC or optically readable bar codes. Sensors are used in objects to collect information, while actuators could be used to manipulate the environments. Localization for smart 'things' could be achieved using GPS, ultrasound time measurement, radio beacons and optical technologies. Creating appropriate user interfaces is another step in ensuring efficient communication and interaction with the user. There are some big challenges that are restricting the growth and proliferation of the Internet of Things. Scalability is a major concern. Communication and service discovery needs to be performed efficiently to ensure IoT reaches its full potential. Interoperability is one of the biggest challenges in the IoT domain. There needs to be a common standardised schema to allow different smart objects with different information, processing and communication capabilities to interact with each other. Another challenge is to create an extensive software infrastructure to manage the 'things' and provide services to them. Other challenges include managing huge volumes of data some 'things' may produce, interpretation of the data, providing security, privacy and fault tolerance. A logical development to the Internet of Things could be to leverage the technologies of World Wide Web for the smart objects as Web of Things [23]. JADE could play a significant role to expedite the development of the different aspects of the Internet of Things and help in overcoming some of the significant challenges. The sensors and actuators provide information to the 'thing', which could transfer this information to the server, by using the 'live' variable concept. Thus, ensuring that frequent updates of the values are sent. JADE also allows to use most of the powerful features of the web, as it allows the developer to write code in JavaScript inside the `'jadef'` function and include external files and libraries by using the `'jarequire'` function. JavaScript is the most widely used and familiar language for the web and with so many new libraries being based on it, JavaScript could prove to be a vital cog in making the development of IoT widespread. The JADE programming framework combined with the NodeJS server which will be implemented soon, and through the use of TCP as the communication protocol and the use of web sockets, could provide the software architecture necessary to allow the 'things' and the physical environment to be controlled virtually.

6.1.1 Machine to Machine Communication

To enable smooth interactions between one machine and another, without human intervention, it is important to overcome the domain specific nature of the existing semantic sensor networks that add semantics to the context. Semantic sensor networks are responsible for enabling explicit representation of sensors, sensor observations and knowledge of the environment. A possible solution is to add semantics to the measured data as opposed to the context [16]. There has been an increase in the availability of M2M devices, due to a growth in the usage of M2M devices in several domains like home monitoring, weather monitoring, health monitoring. M2M area networks gather information from M2M devices. Merging different M2M area networks (sensor networks) to create useful M2M applications is a difficult task due to the differences in the protocols used (Zigbee, Bluetooth, 3G, 4G, WiFi, CoAp, etc.), heterogeneous data formats and the lack of description of measurements. Semantic data in most cases is implicit, however at times there is a need for explicit semantic description of the data. The challenges in this front include managing heterogeneous data from M2M area networks; using semantic web technologies to convert sensor measurement into semantic measurement and the ability to reason on these semantic data. The paper presents an architecture to convert heterogeneous sensor networks to semantic sensor networks. Semantic technologies are added to M2M gateways as well as M2M applications. The heterogeneous nature of the M2M data is overcome by converting all the data into a standard XML format; then using semantic web languages (RDF, RDFS, OWL) to add semantics to the XML sensor measurement. In M2M applications, semantic based reasoning tools (machine earning, recommender system) provide sophisticated semantic treatments. JADE could assist in achieving M2M communication by using peer to peer communication, where a 'thing' broadcasts its information to all the other 'things' in the same space. In addition, a publisher-subscriber scheme could be used; where a 'thing' publishes its events using the `'jaevent'` to the server and the subscriber of the event, indicated using `'jasubscribe'`, is notified when the event gets triggered.

6.1.2 Interoperability

All 'things' should have a common platform for communication, with standard single API. This API should be sustainable over different platforms. Cross domain interoperability, should be given significance over vertical integration. If all smart objects in a space are connected to a single vendor, the problem arises when there is an update in an object that is produced by a different vendor. There are several challenges around interoperability in the web of things (WoT). There is a need to increase interoperability, at the same time maintaining innovation and exploration. There has been an explosion of WoT platforms over the recent years; these often take for granted the presence of an interoperable IoT model. A hub-centric approach [24] is a possible means of overcoming these challenges. WoT hubs can be broken down into a number of categories like web-enabled IoT products, web centric IoT development platforms, WoT hubs and sensor webs. A degree of interoperability is provided through a thing-agnostic model and API created by large IoT platform vendors to allow to integrate 'things' across various domains. As more users and 'things' are connected to the hub, it could become a de facto standard. Domain-specific sensor data portals like International Federation of Digital Seismograph networks (FDSN) have established standards towards interoperability. JADE could help in facilitating interoperability. JADE works using JavaScript as an "interface". That is the native functions (C functions) are exposed as JavaScript functions that are called on "proxy" objects. As JavaScript is a standardised language which is already largely interoperable, we can achieve interoperability using the JADE approach.

6.2 Other Applications

6.2.1 Home Automation

Smart computing is making steady inroads into home automation, where appliances and other items are getting an infusion of computing capabilities. The next step is for the smart objects to operate in concert to support a smarter environment in the home. Interoperability is one of the major challenges in achieving this objective. SpaceOS provides an enabling framework for a smart computing environment by providing a user-driven approach for tackling the interoperability problem. In addition, SpaceOS can also enable newer modes of interactions. In particular, users can employ video to obtain another view of the home environment while controlling the smart devices and interacting with the people in the home.

6.2.2 Remote Monitoring and Control for Elderly Care

The basic setup of this application is very similar to home automation. However, this problem can have stronger privacy issues because of a third party involvement. The hybrid model of overlaying the video stream with smart device information can help. We can decrypt portions of the video that are normally encrypted using keys provided by the smart devices. Depending on the contingency that might arise, the video becomes visible for the care provider.

6.2.3 Physical Devices and Spaces as a Service

There are many valuable physical spaces and devices for which users have limited access. One example would be science laboratories for high school students. A smart computing environment like the one created by SpaceOS can be used to virtualize a laboratory and provide remote access to the facility to a large number of students. The devices in the lab could be connected to microprocessors to make them “smart”. These devices would have a representation on the web browser, which the students could access. Each student could control different objects through the web using this setup created through JADE, as the lab equipments have now have become ‘things’. Further, using the facilities provided by SpaceOS it is possible for large number of users to collaborate in a physical space. In the case of a school project, members of a group could also discuss and have live chats on the web browser, exchange their ideas and then collaborate to work and control different devices to complete their projects remotely. In a project involving robotics, multiple students could work on different parts of a robot by having microprocessors for different parts of the robot. One student could work on the arm, another on the fingers, another on the body.

Chapter 7

Related Work

7.1 End User Programming in Ubiquitous Programming Environment

Ubiquitous programming has been perceived as the third wave of computing for a long time. There have been several academic approaches to realize ubiquitous computing. Based on the extent of human involvement, on one end of the spectrum lies machine learning, which places decision making entirely on the hands of the system. While on the other end is the top down approach to ubiquitous computing called End User Programming, which involves the end user in customizing the environments. End User Programming refers to people who are not professional software developers, programming computers. Various End User Development (EUD) tools could be used by the end user to program and modify softwares without significant knowledge of programming. Some EUD tools include spreadsheets and scripting languages. Although, we have computers everywhere, in phones, cars, TVs, refrigerators and many other devices, there is an evident absence of the software to control these devices and make them work for us. A formal connection lacks between the programming task and its abstraction, and the user of the environment. Domains like mobile devices, browsers, databases, spreadsheets, games et cetera have seen end users embrace programming and customizations. This indicates that end user may be ready to program their ubiquitous environment.

Smart home fitted with actuators and sensors is a commonly cited example of a future ubiquitous computing environment. Readings of the actuators could be coordinated to infer high-level conditions about the state of the home and its occupants. One of the major challenges in building such smart spaces is the massive amount of customization necessary. Due to large and varying individual needs for these applications, the traditional software paradigm will not scale. Requirements to realize this is to have an extensive architecture built on top of existing, well understood web protocols, to allow users to program their smart environment. In the presently implemented models, commercial systems are controlled by the vendor with proprietary rather than open standards. Its a flawed approach, as no single company could provide all the actuators/sensors needed to achieve diverse ubiquitous computing.

A survey was conducted [25] from users about smart homes and their impact. It was apparent from the responses received that different users wanted different behavior from their space; different users perform similar actions in different ways. These varying behaviors and actions could be met by creating abstract, straight forward and usable end user programming constructs for ubiquitous computing. The paper [25] tries to shift the focus to the user and motivate and create a vision for end user programming in a ubiquitous environment. This vision of end user programming follows three major steps. Firstly, user should be involved in the design process. The platform should be human centric and only moderately abstract. Secondly, provide an extensible platform for matching conditions to actions. Performing actions depends on the availability of hardware. Each smart environment is distinct. The paper suggests to use a single software framework, which could be extended to accommodate additional software or hardware functionality. Lastly, to build on powerful, widely deployed web protocol. The platform server could be written on any of the existing web languages.

Implementing ubiquitous programming through JADE framework could achieve most of the desired objectives of End User Programming. For a space which allows Ubiquitous Computing, using the JADE framework, the developer could include the end user during the design process. An extensible platform could be then created, where any new device could be added following similar sequence of events as described when a new device enters a space in SpaceOS, depicted in figures 3.4 and 3.5. Also, JADE uses JavaScript which is the most widely used scripting language for the web and would fit in well with the proposed vision for end user programming to widely deploy web protocols.

7.2 Ubiquitous Computing

In the computing model presented in [26] called Plan B, there is no middleware in place to integrate different systems, instead there is a way of exporting all of the properties of a system through distributed virtual file systems. This Ubiquitous computing model could be used to apply general purpose tools to any system resource without the need of a middleware. All machines are peers and export volumes. System has machines, booted by the user. Once booted, a machine exports all of its resources to the network as a tiny file system. These tiny file systems exported consist of resource volumes and attributes, referred to as Constraints. Constraints are attribute - value pairs to identify properties of interest. Each process has namespace that binds names to resources. The environment that an application could see consists of a set of files that have been imported into its namespace. The environment seen by the user is that seen by all the applications that belong to him/her. Mount system call is used to find the volume given the constraint from the volume table. For handling events, a general purpose event delivery volume provides ports that could be used as event channels. The file system processes event messages and writes to appropriate port to deliver the message. Interoperability between different systems using this model is granted to an extent because most machines nowadays have the ability to remotely use files. As the protocol used by Plan B is not a widely used protocol, some Plan B machines run gateways that export Plan B files through CIFS and NFS. This adds Windows and Unix to the list of systems that could use Plan B files. In contrast to the approach taken by Plan B, JADE uses the cloud server, which is a middleware needed to enable interaction between different 'things'. Through JADE, a 'thing' sends the events that it subscribes for and the events that it publishes to the cloud server, which could be then accessed by other 'things' in the network.

7.3 Programming of Pervasive Computing

The system architecture for pervasive computing makes the developer's task of creating application that adapts to highly dynamic environment, feasible. Pervasive computing calls for the deployment of a wide variety of smart devices that are expected to react to their environment and coordinate with each other and the network services, throughout our living spaces. Pervasive computing space could be envisioned as a combination of mobile and stationary devices that draw on powerful services embedded in the network to achieve users' task. The key challenge is to build applications that adapt to a rapidly evolving environment. Current distributed systems and client server models are not adept to handle such a huge dynamic computing environment. In this architecture called one.world [4], each device runs a single instance of one.world. Separate abstractions exist for application data and for functionality. One.world stores and interacts with data in the form of tuples, and are composed of components. Components implement functionality by importing and exporting event handlers. Each device's root environment holds one.world kernel.

Application needs to be accessible as the user is moving across the physical space. The applications should provide access to shared data even if the current location does not allow network access and should also have the ability to recover from failures. Features like migration, remote event passing (REP), replication and checkpointing are the services provided by one.world that serve as common building blocks to directly help developers to make their applications adaptable.

Data management is achieved through tuples. Tuples are records with named fields and an application that dynamically determines a tuple's fields. Tuples are used for I/O instead of byte stream, because tuples preserve the structure of data, are simple to use and obviate the need for explicit marshalling and unmarshalling of data. Compared to XML, tuple are simpler and easier to use. The structure of XML-based data is more complicated including tags, attributes and namespaces. In addition, the interface to XML-based data such as DOM, are comparatively more complex. Tuples have a Global unique identifier (GUID) field, that supports application specific annotations. Replication makes tuples accessible to multiple nodes even if the tuples are not connected.

Events in one.world are simply tuples. Events have a source field implementing an event handler. Event delivery has at-most-once semantics, both for local and remote event handling. Components import and export asynchronous event handlers and are responsible for implementing application functionality. Components are instantiated within specific environments and within their constructors, declare which event handlers they will import and export. Asynchronous event handling is implemented by using a queue of pending invocations provided by each environment and a pool of one or more threads to implement such invocations. An environment seems like a regular component to an application. There is a hierarchical arrangement of events, which offers considerable flexibility and power. Event sent to the request handler of an event, which is exported by the environment, are delivered to the first ancestral environment whose 'monitor' handler, which is imported by the environment, is linked. One.world is implemented using Java and a small, native library is used to generate the GUID. BerkleyDB [27] is used to implement reliable tuple storage. The one.world architecture for pervasive computing uses three principles to provide system support. Firstly, uses leases to expose changes, allowing the applications to develop strategies to handle those changes. Secondly, nested environments and late binding to dynamically compose applications. Finally, tuples represent data and components implement functionality to cleanly separate data and functionality.

In an analogous manner to the pervasive computing architecture, the JADE framework uses 'live' variables to track and expose changes captured by the sensors. In JADE, we use a publisher-subscriber scheme for binding applications and events, by allowing the developer to specify the attribute of a 'thing' using the `jasubscribe` function. In JADE, the functionality is represented by a JSON object which contains the name of the function while data is represented as arguments. Thus clearly separating data and functionality just like what tuples do in the pervasive computing model presented in [4].

7.4 Olympus: A Pervasive computing programming framework

Olympus is a high-level programming model, developed [28] for Enhanced Physical Spaces called Active Spaces that are highly dynamic. An Active Space is a physically bounded collection such as a room consisting of devices, users, services and applications. Spaces are characterized by large number of different types of services and applications. It's a challenge to choose the "best" way of performing a task, given so many different options for services and applications. The developer should not be burdened with this task, so a higher level of abstraction should be provided to the developers. Olympus allows developers to specify active space entities using high level description. The developer would not know how different tasks are performed in different environments. Each space has different kind of resources, so developers had to customize their application for new spaces. Programs in an active space should choose the "best" way of performing a task, and not bother the developer with this. Olympus is associated with virtual entities, which are resolved by the framework into an actual active space entity. Discovery process discovers a class of entities that satisfy all requirements and then select instances of these classes that satisfy instance level requirements. Execution of common active space operations like starting, stopping and moving components; notifying users; taking actions when the user enters a space, etc. are determined by the Olympus framework and the developer does not need to bother about them. Olympus is part of the Gaia middleware and is implemented in C++. Below is a sample Olympus program. Here the developer says he wants to start a slideshow application with the file Olympus.ppt in an active space:

```
ActiveSpace as1;    // refers to virtual active space entity
as1.instantiate();   // as1 now refers to the active space
Application app1;   // refers to virtual application entity
app1.start(as1);   // app1 started in active space as1
```

Listing 7.1 Example code for Olympus

A meta operating system called Gaia [29], manages resources for an active space. Olympus allows its users to program in terms of virtual entities, which are variables that have not yet been initialized. Entities can be stored in variables, used in expressions and passed as parameters to functions. The developer does not need to worry about the actual instance that the entity gets instantiated with. Commonly used Active space operations are implemented as operators in Olympus. The operator set is analogous to the instruction set of a computer. These could be used by the developers in their programs. For instance, a space may require authentication before a user is allowed to enter, while another may use location service to detect the user's presence. These details are hidden from the developer, and allows him to check the user's presence by the operator 'in'. [30] mentions Gaia to be a distributed middleware infrastructure that coordinates software entities and heterogeneous supported interaction nodes. In most cases using a middleware approach to address pervasive computing, the developer is still left producing the glue code between the middleware and the application domain. Olympus tries to bridge the gap between the midleware and its application domain by enabling ontological description of entities to be integrated into the development of an application. Then depending on different aspects like resource availability and developer-supplied constraints, a middleware based on Gaia resolves the descriptions enabled by Olympus into actual entities. Applications in Gaia are composed of five components, which are model, presentation, adapter, controller and coordinator. The ontology defines the relationship between different objects. For example, there is `requireDevice` relationship. `requireDevice(PowerPointViewer)` = `PlasmaScreen v Desktop v Laptop v TabletPC`. The above example implies that the `PowerPointViewer` features can only run on `PlasmaScreen`, `TabletPC` or `Desktop`. Developer can specify the constraints that the classes and instances of the virtual machine should satisfy in the format of entity, property and values. Space-level policies are written by the administrator of the space in the form of prolog rules. Based on the location, task supported, state of the entity and context of the space, Olympus creates a multidimensional utility function to choose the best entity. As it is not possible to rank entities across dimensions, to rank all candidate entities for choosing the best one, one of the dimensions must be chosen as the primary one. Programs developed on Olympus have two main segments. In the first segment of programming on Olympus, the developer specifies the type of virtual entity and its properties or constraints. In the second programming segment, developers can use high level operators on the entities. Although the framework does simplify the task of the developer, it also takes away some control from the hands of the end user.

JADE has some similarities to Olympus, as it uses known programming languages like C and JavsScript. However, JADE does not implement a lot of abstraction. So, the user and the developer always have the control in their hands, unlike the case in Olympus.

7.5 Spatial Programming

In the world of ubiquitous Networks of Embedded Systems (NES), Spatial Programming (SP) [31] attempts to solve problems that are faced by traditional distributed programming. It's an attempt to design and implement a programming model for NES, which is able to execute coordinated actions in a decentralized manner. Space is split into two special spaces, space1 and space2. Space1 is a given geographic location and space2 is detected dynamically by using an intelligent camera, once a motion sensor is triggered by an object. The main focus is to program the physical world in order to overcome the challenges posed by attempting to execute coordinated actions in a decentralized manner. So, when a motion sensor is triggered by an object, collaborative object tracking needs to be performed by the cameras. Spatial reference is provided by using a tuple `{space:tag}`. The drawback of this approach is that the namespaces, to reference the space, could be extremely large.

7.6 SpaceBrew

Spacebrew [32] is an MIT licensed open software toolkit to connect interactive things to one another. Every element that is hooked up to the system is identified as either a publisher or a subscriber. To launch Spacebrew, a server is needed to host the

Spacebrew session. Public Spacebrew session could be hosted on Amazon EC2 as well. The public server is written in NodeJS and uses web sockets. Any web browser based client can communicate with the Spacebrew. When connecting to the server, clients need to communicate their 'config' information to Spacebrew. In order to take the html/css/javascript coding out of the web side of things, Spacebrew consists of a series of libraries for Arduino, Processing, JavaScript. Thus reducing the knowledge of web technologies to prototype projects that communicate via the internet. Whenever a Spacebrew example is run (processing, javascript etc.), a "publisher" or a "subscriber" is created on the spacebrew server. The interface visible to the user consists of three columns, which are publisher, subscriber and clients. A connected client sending out some data is called a "publisher" and appears on the left column of the server. A client that receives and interprets the information is called a "subscriber" and appears on the right column of the server. Initially, each client on making a connection with Spacebrew, appears in the clients column. Boolean, ranges and strings are the types that the client is capable of sending; they appear as nodes in the publisher column. Information that client could interpret appear as nodes in the subscriber column. The user could hover over and connect like values (Boolean to Boolean, string to string, publisher to subscriber) between publishers and subscribers. Below is a code sample written in processing.js

```
c = new Spacebrew( this );
// add each thing you publish
// and subscribe to
c.addPublish( ``buttonPress", buttonSend );
c.addSubscribe( ``color", ``range" );
```

```
c.addSubscribe( ‘‘text”, ‘‘string” );
```

Listing 7.2 Spacebrew syntax

JADE has some similarities to Spacebrew. JADE also uses a publisher-subscriber scheme. For the subscriber of an event we could use ‘jasubscribe’ with the name of the event, which is explained in Chapter 4. The keyword ‘jaevent’ could be used to publish the events. Like Spacebrew, JADE also tries to make the work of the developer and the user easier, by hiding the underneath complex architecture for sending data to the server and reusing existing technologies. The primary difference is the customizable interface offered by JADE. Spacebrew offers a ‘standard’ interface like a service-oriented system.

Chapter 8

Conclusions and Future Work

Internet is poised to undergo a major transformation in terms of number of devices and services due to the induction of Internet of things (IoT). To fully realize the promise of IoT, we need programming frameworks that will allow developers to work on IoT with familiar tools. To achieve this, we propose JADE that constructs a programming framework using C/C++ and JavaScript. The ideas used in this synthesis could be applied to create a version that mixes Java with JavaScript as well. We believe the approach of using a preprocessing step and leaving the actual compilation to the "native" compiler has its benefits in long term maintenance of the tools and the application code.

To further simplify the application development for IoT, we introduced the ideas of 'live' variables. Although the functionality offered by the live variables can be implemented by the event processing scheme in JADE, we believe the live variables provided a much easier usage pattern for the programmers.

Another aspect of JADE that is yet to be implemented is the virtualization of the things. When a thing could be controlled by many users over the web, it becomes complicated for the application developer. We believe the JADE runtime could actually help to make the programmers' task easier. By virtualizing the 'thing', we allow the programmer to deal with the single user scenario and the runtime is responsible for mapping the actions in the multi-user scenario.

We already built the proof-of-concept prototype of the JADE preprocessor and runtime. We are able to write programs and execute them.

As part of the future work, we will implement the 'thing' virtualiztion. We will use the 'thing' virtualization to deploy JADE applications over many web users (i.e., multi-user scenario). Another interesting future direction is to use JADE in newer operating systems such as Tizen that have the "web app stack" and "native app stack." Using JADE, we could develop "hybrid apps" that use the web and native features in a single device or across multiple devices. Also as part of the future work, we will try to implement some of the methods described in Chapter 3 for achieving reliability and a higher level of security.

The server in the current version of spaceOS is implemented in python using the Tornado framework. In the future we intend to implement the server in NodeJS. This will allow the `jadef` function for the server to directly introduce parts of javascript code to the server.

The JADE file creates a Javascript file and a C / .ino file. The JavaScript file needs to be sent to the server, so that the JavaScript could be rendered on the browser, when the space is accessed. This could be achieved by using a File transfer protocol.

There is a growing trend for applications to be developed which are part native and part web based. Different devices use different browser technology for rendering web pages. Blackberry uses HTML5WebWorks for development, iOS devices use

PhoneGap while smart watches use the Tizen Web API.

References

[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future Gener. Comput. Syst.*, vol. 29, pp. 1645–1660, Sept. 2013.

[2] S. Nastic, S. Sehic, M. Vgler, H. L. Truong, and S. Dustdar, "Patricia - a novel programming model for iot applications on cloud platforms.," in *Service-Oriented Computing and Applications*, pp. 53–60, IEEE, 2013.

[3] D. Guinard, *A Web of Things Application Architecture – Integrating the Real-World into the Web.* Ph.d., ETH Zurich, 2011.

[4] R. Grimm, J. Davis, E. Lemar, A. Macbeth, S. Swanson, S. Gribble, T. Anderson, B. Bershad, G. Borriello, and D. Wetherall, "Programming for pervasive computing environments," tech. rep., University of Washington, 2001.

[5] R.-C. Marin, "Hybrid contextual cloud in ubiquitous platforms comprising of smartphones," *Int. J. Intell. Syst. Technol. Appl.*, vol. 12, pp. 4–17, July 2013.

[6] A. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39–59, 1984.

[7] J. Maassen, R. V. Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman, "Efficient java rmi for parallel programming," *ACM Transactions on Programming Languages and Systems*, vol. 23, p. 2001, 2001.

[8] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple Object Access Protocol (SOAP) 1.1," w3c note, World Wide Web Consortium, May 2000. See `http://www.w3.org/TR/SOAP/`.

[9] K. A. Kadouh and K. A. Albashiri, "Improvement of data transfer over simple object access protocol (soap)," *International Journal of Computer, Information Science and Engineering*, vol. 8, no. 2, pp. 16 – 19, 2014.

[10] R. T. Fielding, *REST: Architectural Styles and the Design of Network-based Software Architectures.* Doctoral dissertation, University of California, Irvine, 2000.

[11] B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao, "Tiny web services for sensor device interoperability.," in *IPSN*, pp. 567–568, IEEE Computer Society, 2008.

[12] "BEA, microsoft, and tibco release web services eventing (WS-eventing) specification.," 2004.

[13] D. J. Cook, J. C. Augusto, and V. R. Jakkula, "Review: Ambient intelligence: Technologies, applications, and opportunities," *Pervasive Mob. Comput.*, vol. 5, pp. 277–298, Aug. 2009.

[14] D. Preuveneers, J. V. den Bergh, D. Wagelaar, A. Georges, P. Rigole, T. Clerckx, Y. Berbers, K. Coninx, V. Jonckers, and K. D. Bosschere, "Towards an extensible context ontology for ambient intelligence," in *Second European Symposium on Ambient Intelligence*, vol. 3295 of *LNCS*, pp. 148 – 159, Nov 8 – 11 2004.

[15] R. Rajkumar, I. Lee, L. Sha, and J. A. Stankovic, "Cyber-physical systems: the next computing revolution.," in *Design Automation Conference* (S. S. Sapatnekar, ed.), pp. 731–736, ACM, 2010.

[16] A. Gyrard, C. Bonnet, and K. Boudaoud, "A machine-to-machine architecture to merge semantic sensor measurements," in *WWW 2013, 22nd International World Wide Web Conference, Doctoral Consortium, May 13-17, 2013, Rio de Janeiro, Brazil.*

[17] R. Saracco, "Future of objects virtualization and the internet with things," in *IEEE Technology Time Machine Symposium on Technologies Beyond 2020*, pp. 1–1, June 2011.

[18] D. Kelaidonis, A. Somov, V. Foteinos, G. Poulios, V. Stavroulaki, P. Vlacheas, P. Demestichas, A. Baranov, A. Biswas, and R. Giaffreda, "Virtualization and cognitive management of real world objects in the internet of things," in *IEEE International Conference on Green Computing and Communications*, pp. 187–194, Nov 2012.

[19] "Introducing the intel galileo development board." See `http://www.intel.com/content/www/us/en/do-it-yourself/galileo-maker-quark-board.html`.

[20] "Triple axis accelerometer breakout - mma8452q - sen-10955 - sparkfun electronics." See `https://www.sparkfun.com/products/10955`.

[21] D. Crockford, "Javascript object notation.," See `http://www.json.org/`.

[22] F. Mattern and C. Floerkemeier, *From the Internet of Computers to the Internet of Things*, vol. 6462 of *Lecture Notes in Computer Science*, pp. 242–259. Springer, 2010.

[23] D. Guinard and V. Trifa, "Towards the web of things: Web mashups for embedded devices," in *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences)*, Apr. 2009.

[24] M. Blackstock and R. Lea, "Toward interoperability in a web of things.," in *UbiComp (Adjunct Publication)* (F. Mattern, S. Santini, J. F. Canny, M. Langheinrich, and J. Rekimoto, eds.), pp. 1565–1574, ACM, 2013.

[25] S. Holloway and C. Julien, "The case for end-user programming of ubiquitous computing environments," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pp. 167–172, ACM, 2010.

[26] F. J. Ballesteros, E. Soriano, K. L. Algara, and G. G. Muzquiz, "Plan b: An operating system for ubiquitous computing environments.," in *PerCom*, pp. 126–135, IEEE Computer Society, 2006.

[27] M. A. Olson, K. Bostic, and M. I. Seltzer, "Berkeley db.," in *USENIX Annual Technical Conference, FREENIX Track*, pp. 183–191, USENIX, 1999.

[28] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. D. Mickunas, "Olympus: A high-level programming model for pervasive computing environments.," in *PerCom*, pp. 7–16, IEEE Computer Society, 2005.

[29] M. Romn, C. Hess, R. Cerqueira, R. H. Campbell, and K. Nahrstedt, "Gaia: A middleware infrastructure to enable active spaces," *IEEE Pervasive Computing*, vol. 1, pp. 74–83, 2002.

[30] C. Consel, W. Jouve, J. Lancia, and N. Palix, "Ontology-directed generation of frameworks for pervasive service development.," in *PerCom Workshops*, pp. 501–508, IEEE Computer Society, 2007.

[31] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode, "Spatial programming using smart messages: Design and implementation.," in *International Conference on Distributed Computing Systems*, pp. 690–699, IEEE Computer Society, 2004.

[32] "About spacebrew." See `http://docs.spacebrew.cc/about/`.