JAMScript – A Programming Framework for Cloud of Things

Jayanth Krishnamurthy

School of Computer Science
McGill University
Montreal, Canada

May 2016

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

Acknowledgments

This thesis wouldn't have been completed without the guidance and mentoring from my thesis advisor, help from friends, and support from my family.

This thesis is a part of an ongoing project which is the brainchild of my advisor. My deepest gratitude to Professor Muthucumaru Maheswaran. I have been very fortunate to have a mentor and guide like Professor Maheswaran, who gave me freedom in the assigned work to explore on my own, and at the same time the guidance to correct whenever I faltered. I thank him for his time, patience, technical insight, and for including me in this project. Further, I also thank my research group at ANRL lab, McGill for their continuous support and motivation.

I take this opportunity to thank my employer back in India, for granting me study leave to pursue my graduate studies. But for the support, sacrifice of my parents and my younger brother, I wouldn't have come so far. I'am deeply indebted to them.

Last but not the least, I thank the IoT-Cloud research community for their continuous pursuit in developing better programming frameworks to develop new age applications.

Abstract

Cloud of Things (CoT) is a new computing paradigm that combines the widely popular Cloud computing with Internet of Things (IoT). Programming CoT brings many interesting challenges as clouds and things have varying capabilities and responsibilities. They are expected to play their predetermined roles even in the combined programming model. In a typical deployment, the Cloud is responsible for heavy data processing operations and long-term, huge data storage; while the things are responsible for sensing data and actuating the control signals from the cloud. In this thesis, we present the design and implementation of a new programming paradigm, "JAMScript", that combines the hugely popular C and JavaScript in an unique distributed computing model that can support both parallel and concurrent computations. The objective of JAMScript is to allow the developers to exploit the heterogeneity of CoT while providing support for fault tolerance and low overhead computing. JAMScript simplifies the task of integrating legacy embedded C programs to the cloud with minimal coding efforts. Also, in this thesis, we review many of the technologies, programming models that can help CoT programming and present a detailed survey of various IoT/CoT frameworks that have been developed recently.

Résumé

Cloud of Things (TCO) est un nouveau paradigme informatique qui combine le Cloud computing très populaire avec l'Internet des objets (IdO). CoT Programmation apporte de nombreux défis intéressants que le programme Cloud of things ont différentes capacités et les responsabilités. Ils sont appelés à jouer leurs rôles prédéterminés même dans le modèle de programmation combinée. Dans un déploiement typique, le Cloud est responsable des opérations de traitement de données lourds et à long terme, de stockage énorme de données ; tandis que les choses sont responsables pour détecter les données et actionner les signaux de commande à partir du nuage. Dans cette thèse, nous présentons la conception et la mise en œuvre d'une nouvelle programmation paradigme, quot ;'JAMScript', qui combine la C très populaire et JavaScript dans un modèle unique de calcul distribué qui peut soutenir les deux calculs parallèles et concurrentes. L'objectif de JAMScript est de permettre aux développeurs d'exploiter l'hétérogénéité des CoT tout en fournissant un soutien pour la tolérance aux pannes et une faible surcharge informatique. JAMScript simplifie l'intégration task of programmes hérités embarqué C vers le cloud avec les efforts de codage minimal. Aussi, dans cette thèse, nous passons en revue un grand nombre des technologies, des modèles qui peuvent aider à la programmation de lit et présenter une étude détaillée des différents cadres / COT IdO qui ont été récemment mis au point la programmation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The development of smart environments [1] is happening in at least three phases. The first phase is the infusion of computing capabilities into everyday objects such as doorknobs, lamps, and thermostats. The second phase is to interconnect the smart objects using standardized protocols. Cisco[1] has estimated that at least 50 billion devices will be Internet enabled by the year 2020. The technological challenges posed by these two phases are tackled by various initiatives such as Internet of Things [2], Web of Things [3], and IoTivity[2], Cloud of Things (CoT) [4]. The third phase is the development of novel operating systems [5, 6] and programming frameworks [7] that will provide a single system image [8] over the disparate collection of things that makes up a smart environment. The attributes of the smart environments provide many opportunities and challenges for the development of a new computing paradigm.

IoT devices are generally characterized as small things in real world with limited storage and processing capacity, which may not be capable of processing a complete computing activity by themselves. They may need the computational capabilities of Cloud based back-ends to complete the processing tasks and web based front-ends to interact with the user. The Cloud infrastructure complements the things [9], by supporting device virtualization, availability, provisioning of resources, support-

[1]http://www.cisco.com/web/solutions/trends/iot/portfolio.html
[2]https://www.iotivity.org/

ing data storage and performing data analytics. The IoT by its nature will extend the scope of Cloud computing to the real world in a more distributed and dynamic way [10]. The integration of Cloud computing and things, can also be called as Cloud of Things (CoT). IoT with Cloud – CoT, will create new avenues for computing: huge storage capacity for IoT data in cloud, massive computing capabilities to collect, analyze, process, archive those data, and new platforms like SaaS (Sensing as a Service), SAaaS (Sensing and Actuation as a Service), VSaaS (Video Surveillance as a Service) will open up to users. Throughout the world, many governments and organizations are investing a significant amount of their GDP and profits on projects like smart cities, smart roads, smart buildings, smart health care, etc,. These multi-billion dollar projects would require huge support by CoT [10] to have an effective implementation. The biggest challenge for applications development in CoT ecosystem is to have a single, unified programming framework for both Cloud and things that can simplify the developers' role and hide many of the low level issues like communication, heterogeneity, and fault tolerance from the programmer. The C language and JavaScript have been the most popular languages among the programming community and also have their own advantages in embedded programming and web application development. We merge these two under a novel programming framework called JAMScript for CoT application development.

1.1 Motivation

The challenges of developing applications for IoT/CoT serves as the motivation for our framework. Heterogeneity and the volume of data generated are two of the biggest concerns. Heterogeneity spans through hardware, software and communication platforms. The data generated from these devices are generally in huge volume, are in various forms and are generated at varying speeds. Since CoT applications will be distributed over a wide and varying geographical area, support for corrective and evolutionary maintenance of applications will determine the feasibility of applications deployment. Further, some of the CoT applications like traffic management system, will be latency sensitive and this warrants edge-processing support by the

programming framework. Another difficulty faced when programming CoT, is how to cope with frequent periods of non-availability of devices caused due to mobilities and limited energy supplies from batteries. There are already huge number of embedded devices running standard applications written in languages like C/C++; If these devices are empowered to connect via Internet, it opens up various applications to a wider audience.

Developing a simplified programming model that can provide solutions for the above set of challenges will remain a continuous pursuit for IoT/CoT community. JAMScript helps in integrating embedded applications on to Cloud in a simple, fault tolerant, and secure manner. We believe JAMScript will be more advantageous and cost effective for the new age applications development.

1.2 Thesis Contribution

The contribution of this thesis is three fold – the main contribution, is a novel programming framework for CoT, "JAMScript". It mainly consists of a new coordination language [11] and a runtime for supporting its execution [12]. Second – a detailed survey on various IoT/CoT programming frameworks and approaches which have been developed recently. Many of the features of the JAMScript design are inspired by the evolving technologies and prevailing programming models; a detailed review of those forms the third component of this thesis.

1.3 Thesis Organization

In Chapter 2, we discuss the background technologies and programming paradigms that have inspired our framework design. The design of JAMScript framework is explained in Chapter 3, Chapter 4 explains the implementation aspects and the current status of the prototype. Chapter 5 discusses possible application scenarios. Related work is discussed as a survey of various IoT/CoT programming frameworks in Chapter 6, Chapter 7 concludes the thesis.

Chapter 2

Background

2.1 Overview

During the life-cycle of CoT applications, the footprint of an application and the cost of its language runtime play a huge role on the sustainability of an application. C has been used predominantly in embedded applications development due to its performance and it can occupy the same position in CoT programming too. Further, the choice of communication protocols also has a huge implication on the cost of CoT applications on devices. Remote Procedure Calls (RPC), Representational state transfer (REST), and Constrained Application Protocol (CoAP) are some of the communication methods that are being currently incorporated into CoT communication stacks. A complete programming framework in a distributed environment requires not only a stable computing language like C, but also a coordination language that can manage communications between various components of an CoT ecosystem. An explicit coordination language can tackle many of the challenges. It can manage communication between heterogeneous devices, coordinate interaction with the Cloud and devices, handle asynchronous data arrival and also can provide support for fault tolerance. The method of using more than one language in a given application is known as polyglot programming. Polyglot programming is being widely used in web applications development and it can provide the same advantages

for CoT programming too.

In this chapter, we review some of the flavors of C language used in embedded programming, check adoptability of messaging approaches such as RPC, REST, and CoAP to CoT, explore some of the important features of various coordination languages and in the last part of this section, we present the idea of polyglot programming.

2.2 Embedded Devices Programming Languages

Though there are various programming languages in the embedded programming domain, vast majority of projects, about 80%, are implemented in C and its flavors or a combination of C and other languages like C++ [13]. Some of the striking features of C that aid in embedded development are performance, small memory foot print, access to low level hardware, availability of large number of trained/experienced C programmers, short learning curve, and compiler support for vast majority of devices [14]. The ANSI C standard provides customized support for embedded programming. Many embedded C compilers based on ANSI C usually:

1. Support low level coding to exploit the underlying hardware.
2. Support for in-line assembly code.
3. Flag dynamic memory allocation and recursion.
4. Provide exclusive access to I/O registers.
5. Support accessing registers through memory pointers.
6. Allow bit level access.

   nesC, Keil C, Dynamic C and B# are some of the flavors of C used in embedded programming.

2.2.1 nesC

nesC [15] is a dialect of C that has been used predominantly in sensor nodes programming. It was designed to implement TinyOS [16], an operating system for sensor networks. It is also used to develop embedded applications and libraries. In nesC, an application is a combination of scheduler and components wired together by specialized mapping constructs. nesC extends C through a set of new keywords. To improve reliability and optimization nesC programs are subject to whole program analysis and optimization at compile time. nesC prohibits many features that hinder static analysis like function pointers and dynamic memory allocation. Since nesC programs will not have indirections, call-graph is known fully at compile time, aiding in optimized code generation.

2.2.2 Keil C

Keil C [17] is a widely used programming language for embedded devices. It has added some key features to ANSI C to make it more suitable for embedded devices programming. To optimize storage requirements, three types of memory models are available for programmers: small, compact, and large. New keywords like `alien`, `interrupt`, `bit`, `data`, `xdata`, `reentrant`, etc., are added to the traditional C keyword set. Keil C supports two types of pointers:

- generic pointers: can access any variable regardless of its location.
- memory specific pointers: used to access variables stored in data memory.

The memory-specific pointers based code execute faster than the equivalent code using generic pointers. This is due to the fact that the compilers can optimize the memory access; since the memory area accessed by pointers is known at compile time.

Functions in Keil C

The function declarations in Keil C is quite interesting and has many options, the programmer can specify. The general format is as follows.

```
[return_type]Function_name([arguments])[memory_model] [reentrant][
    interrupt n][using n]{}
```

Listing 2.1 – function format in keil C

Re-entrant functions can be called recursively and simultaneously by two or more processes; a re-entrant stack is associated with each of these functions.

2.2.3 Dynamic C

Some key features in Dynamic C [18], are function chaining and co-operative multitasking. Segments of code can be distributed in one or more functions through function chaining. Whenever a function chain executes, all the segments belonging to that particular chain execute. Function chains can be used to perform data initialization, data recovery and other kinds of special tasks as desired by the programmer. The language provides two directives `makechain`, `funcchain` and a keyword `segchain` to manage and define function chains.

`#makechain chain_name`: creates a function chain by the given name.

`#funcchain chain_name func_name[chain_name]`: Adds a function or another function chain to a function chain.

`segchain chain_name {statements}`: This is used for function chain definitions. The program segment enclosed under curly braces will be attached to the named function chain.

The language stipulates `segchain` definitions to appear immediately after data declarations and before executable statements as shown in the following code snippet.

```
int foo(){
// data declarations
segchain recover{
// some statements which execute under function chain recover().
}
segchain chain_x{
// some statements which execute under function chain chain_x().
}
// function body of foo.
```

```
}
int foo1(){
// data declarations
segchain recover{
// some statements which execute under function chain recover().
}
// function body of foo1.
}
```

Calling a function chain inside a program is similar to calling a void function that has no parameters.

```
int foo2(){
.....
.....
recover()/* executes all the statements defined under
function chain recover */
}
```

The order of execution of statements inside a function chain is not guaranteed. Dynamic C's `costate`, statement provides support for co-operative multitasking. It provides multiple threads of control, through independent program counters that can be switched in between explicitly. The following code snippet is an example.

```
for(;;){
    costate{
        waitfor(tcp_packet_port_21());
        yield; // force context switch.
        ...
        }
    costate{
        waitfor(tcp_packet_port_23());
        }
}
```

The `yield` statement immediately passes control to another `costate` segment. If the control returns to the first `costate` segment, then the execution resumes from the statement following the `yield` statement. Dynamic C also has keywords: `shared`

and `protected`, which support data that are shared between different contexts and are stored in battery-backed memory, respectively.

2.2.4 B#

B#[1] is a multi-threaded programming language designed for constrained systems. Though C inspires it, its features are derived from a host of languages like Java, C++, and C#. It supports object oriented programming. The idea of boxing/unboxing conversions is from C#. For example, a float value can be converted to an object and back to float as shown in the following code snippet.

```
class test{
    static void main(){
                        float i = 123;
                        object obj = i; // boxing
                        float j = (float)obj; // Unboxing
                    }
}
```

The field property is also similar to C#. B# provides support for multi-threading and synchronization through `lock` and `start` statements, which are similar to `when` and `cobegin`, from Edison [19]. `lock` provides mutual exclusion and synchronization support, while, `start` is used to initiate threads. Other important features are device addressing registers and explicit support for interrupt handlers. These features are directly supported by the underlying Embedded Virtual Machine (EVM) which interprets and executes the binary code generated by the B# assembler on a stack based machine. The B# EVM runs on a target architecture, thereby hiding the hardware nuances from the programmer. Presence of EVM promotes reusability of components. Also, since the EVM is based on the stack machine model, the code size is much reduced. The EVM also has a small kernel for managing threads.

All the above languages have been optimized for resource constrained devices. While designing embedded programs, a measured choice on the flavor of C, is quite an important decision from the point of an CoT programmer. An CoT programmer

[1] http://www.bsharplanguage.org/

may not restrict him/her to a C-flavored language. Many other languages like C++, Java, and JavaScript have been stripped down to run on embedded devices.

2.3 Message Passing in Devices

In this section, we review some of the communication paradigms and technologies like RPC, REST, and CoAP that can be used in resource constrained environments.

2.3.1 RPC

RPC [20] is an abstraction for procedural calls across languages, platforms, and protection mechanisms. For CoT, RPC can support communication between devices, as it implements the request/response communication pattern. Typical RPC calls exhibit synchronous behavior. When RPC messages are transported over the network, all the parameters are serialized into a sequence of bytes. Since serialization of primitive data types is a simple concatenation of individual bytes, the serialization of complex data structures and objects is often tightly coupled to platforms and programming languages [21]. This strongly hinders the applicability of RPCs in CoT due to interoperability concerns.

Lightweight Remote Procedure Call (LRPC) [22] was designed for optimized communication between protection domains in the same machine, but not across machines. Embedded RPC (ERPC) in Marionette [23], uses a fat-client like PC and thin-servers like nodes architecture. This allows resource rich clients to directly call functions on applications in embedded devices. It provides poke and peek commands that can be used on any variables in a node's heap. S-RPC [21], is another lightweight remote procedure calls for heterogeneous WSN networks. S-RPC tries to minimize the resource requirements for encoding/decoding and data buffering. A trade-off is achieved based on the data types supported and their resource consumption. Also, a new data representation scheme is defined which minimizes the overhead on packets. A lightweight RPC has been incorporated into TinyOS, nesC [24] environment. This approach promises ease of use, lightweight implementation, local call semantics, and

adaptability.

2.3.2 REST

Roy Fielding in his PhD thesis [25] proposed the idea of RESTful interaction for the web. The main aim of the REST was to simplify the web application development and interaction. It leverages on the tools available on Internet and stipulates the following constraints on application development:

1. Should be based on client-server architecture and the servers should be stateless.
2. Support should be provided for caching at the client side.
3. The interface to servers should be generic and standardized (URI).
4. Layering in the application architecture should be supported and each of the layers shall be independent.
5. Optional code-on demand should be extended to clients having the capability.

These constraints combined with the following principles define the RESTful approach to application development.

1. Everything on the Internet is a resource.
2. Unique identifiers are available to identify the resources.
3. Generic and simple interfaces are available to work with those resources.
4. Communication between client and servers can be through representation of resources.
5. Resource representation through sequence of bytes followed by some meta data explaining the organization of the data.
6. Since transactions are stateless, all interactions should be context free.

7. Layering is supported and hence intermediaries should be transparent.

The authors in [26] have highlighted that the above constraints and principles bring in many advantages to the distributed applications: scalability, loose coupling, better security, simple addressability, connectedness, and performance. Further, they compare RPC with REST, for the same qualitative measures and argue that RESTful approaches are always better for each of the above measures. One more advantage of RESTful components is that they can be composed to produce mashups, giving raise to new components which are also RESTful. In [27] the author identifies essential characteristic features of a composing language that can compose RESTful components together:

1. Support for dynamic and late binding;
2. Uniform interface support for composed resource manipulation;
3. Support for dynamic typing;
4. Support for content type negotiation;
5. Support for state inspection of compositions by the client.

Though the uniform interface constraint promotes scalability by shifting the variability from interface to resource representation, it also narrows the focus of RESTful approaches to data and its representations. Also, in the Internet, the exchanges need not be limited to data and its representation; there can be more than, just the pure data. For these cases, the optional code-on demand constraint for clients has been found to be inadequate for exchanges other than content. Also, the RESTful approach poses a challenge for those applications that require stateful interactions.

CREST (Computational REST) [28] tries to address these problems. Here, the focus is on exchanges of computation rather than on data exchange. Instead of client-server nomenclature, everyone is addressed as peers; some may be strong and some may be weak based upon the available computing power. Functional languages like Scheme, allow computations to be suspended at a point and encapsulated as a single

entity to be resumed at a later point of time, through "continuation". CREST's focus is on these sort of computations. It supports the model of "computations stopping at a point in a node, exchanged with another node, resumed from the suspended point at the new node". As said earlier, both the nodes are peers. CREST has some principles along the lines of REST:

1. All computations are resources and are uniquely identified.
2. Representation of resources through expressions and metadata.
3. All computations are context-free.
4. Support for layering and transparent intermediaries.
5. All the computations should be included inside HTTP.
6. Computations can produce different result at different time.
7. Between calls they can maintain states that may aid computations like aggregation.
8. Between different calls, computations should support independency.
9. Parallel synchronous invocations should not corrupt data.

Computations on a peer or on different peers can be composed to create mashups. Peers can share the load of computations to promote scaling and latency sensitive applications.

2.3.3 CoAP

Since HTTP/TCP stack is known to be resource demanding on constrained devices, protocols like Embedded Binary HTTP (EBHTTP), Compressed HTTP Over PAN (CHoPAN) have been proposed. However, the issue of reliable communications still remained a concern. The IETF work group: Constrained RESTful Environments (CoRE) has developed a new web transfer protocol called Constrained Application

Protocol (CoAP), which is optimized for constrained power and processing capabilities of CoT. Although, the protocol is still under standardization, various implementations are in use. CoAP in simpler terms is a two-layered protocol. A messages layer, interacting with the UDP and another layer for request/response interactions using methods and response codes, as done in HTTP. In contrast to HTTP, CoAP exchanges messages asynchronously and uses UDP.

The CoAP has four types of messages: Acknowledgement, Reset, Confirmable (CON), and Non-Confirmable (NON). The Non-confirmable messages are used to allow sending requests that may not require reliability. Reliability is provided by the message layer and will be activated when Confirmable messages are used. The Request methods are: GET, POST, PUT and DELETE of HTTP. CoAP has been implemented on Contiki [29], which is an operating system for sensor networks and in TinyOS as Tiny-CoAP [30].

Many approaches have been used to evaluate the performance of CoAP. Total Cost of Ownership (TCO) model for applications on constrained environment has been used to compare HTTP versus CoAP [31]. The major observations from the comparison are as follows.

- CoAP is more efficient for applications on smart object, engaged in frequent communication sessions.
- CoAP is cost-effective whenever the battery/power source replacements prove costly.
- Whenever the charges for the data communication is volume based, CoAP is found to be more cost effective.
- Also, CoAP has been found to be beneficial cost wise in push mode than in pull mode.

Figure 2.1 illustrates the CoAP layers and the integration of constrained devices using CoAP with the Internet through a proxy.

For CoT, the advantages of CoAP can be summarised as follows.

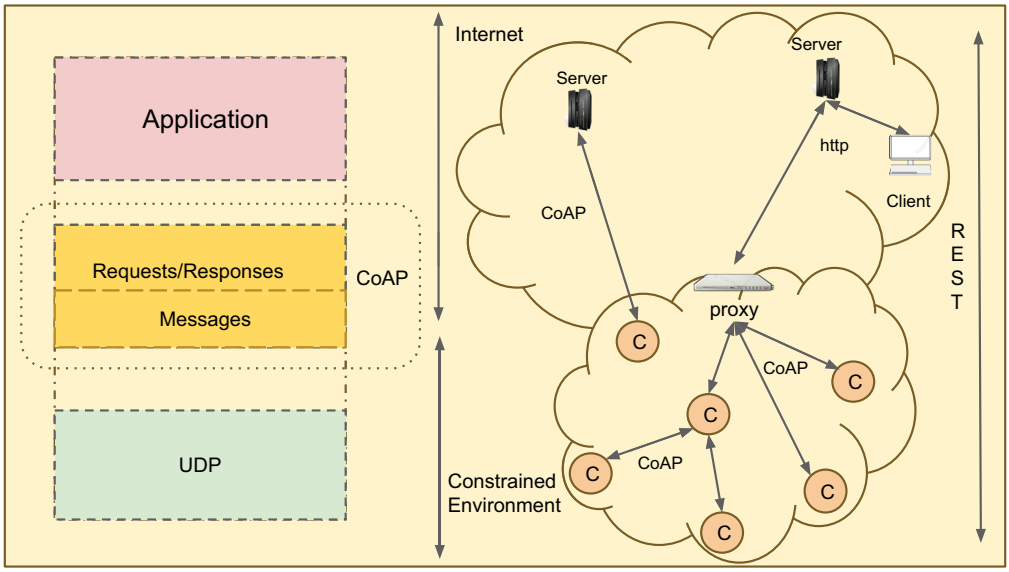


Figure 2.1 – CoAP layers and integration of constrained devices with the Internet.

- A compact binary header (10-20 bytes), along with UDP, reduces the communication overhead; thereby reducing the delay and minimizing the power consumption due to data transmission.
- Since asynchronous data push is supported, it enables things to send information only when there is a change of observed state. This allows the things to sleep most of the time and conserve power.
- The minimal subset of REST requests supported by CoAP, allows the protocol implementations to be less complex when compared to HTTP. This lowers the hardware requirements for the smart things on which it executes.
- The M2M resource discovery is supported by CoAP to find matching resource based on the CoRE link format.
- The draft CoAP proposal includes support for alternative non-IP messaging, such as Short Message Services (SMS) and transportation of CoAP messages over Bluetooth, ZigBee, Z wave, etc,.

MQ Telemetry Transport (MQTT)[2] protocol is another communication protocol designed for M2M communication, based on TCP/IP. Both CoAP and MQTT are expected to be widely used in CoT communication infrastructure in the future.

2.4 Coordination Languages

Carriero and Gelernter argue in [12] that a complete programming model can be built by combining two orthogonal models – a computation model and a coordination model. The computation model provides the computational infrastructure and programmers can build computational activity using them, whereas the co-ordination model provides the support for binding all those computational activities together. They argue that a computational model supported by languages like C, by themselves cannot provide genuine co-ordination support between various computing activities. This observation is more relevant in CoT programming wherein there are numerous distributed activities, which have to be coordinated in a reliable and fault tolerant manner.

Coordination can be seen through two different perspectives i) based on centralised control named as Orchestration and ii) distributed transparent control named as Choreography. The W3C's Web services choreography working group defines Choreography as "the definition of the sequences and conditions under which multiple cooperating independent agents exchange messages in order to perform a task to achieve a goal state". Orchestration is seen as "the definition of sequence and conditions in which one single agent invokes other agents in order to realize some useful function". There are many languages that provide Choreography and Orchestration support. We briefly review some of the features in coordination languages like Linda, eLinda, Orc and Jolie.

[2]http://mqtt.org/

2.4.1 Linda and eLinda

Linda is a coordination-programming model for writing parallel and distributed applications. It takes the responsibility of enforcing communication and coordination, while general purpose languages like C, C++, Java are used for computational requirements of the application. The Linda model supports a shared memory store called tuple space for communication between processes of the application. Tuple spaces can be accessed by simple operations like "out" and "in". These operations can be blocking or non-blocking. CppLINDA is a C++ implementation of the Linda coordination model.

The eLinda [32] model extends Linda. It adds a new output operation "wr" that can be used with the "rd" input operation to support broadcast communication. In Linda, if a minimum value of a data set stored in a tuple space is required, all matching field values should be read, the reduction should be performed and then the remaining data should be returned to the tuple space. While this procedure is accessing the tuple space to extract the minimum value, the tuple space is not accessible to other processes, which restricts the degree of parallelism by a great amount. eLinda proposes the "Programmable Matching Engine" (PME) to solve problems like the above. The PME allows the programmer to specify a custom matcher that can be used internally to retrieve tuples from the shared store. The PME has been found to be advantageous for parsing graphical languages and video-on-demand systems.

2.4.2 Orc

Orc [33] is a coordination language for distributed and concurrent programming. It is based on process calculus. It provides uniform access to computational services, including distributed communication and data manipulation. A brief overview of the language features is as follows:

- The basic unit of computation in Orc is called a *site*, similar to a function or a procedure in other languages. The sites can be remote and unreliable.

- Sites can be called in the form of C(p), C is a site name and p is the list of parameters. The execution of a site call invokes the service associated with the site. The call publishes the response, if the site responds.

- Orc has the following combinator-operators to support various compositions and work-flow patterns [34].

  - Parallel combinator "|" is used for parallel, independent invocation. For example, in `I | J`, expressions `I` and `J` are initiated at the same time independently. The sites called by `I` and `J` are the ones called by `I | J` and any value published by either `I` or `J` is published by `I | J`. There is no direct interaction or communication between these two computations.

  - Sequential combinator "»" is used for invocations of sites in a sequential manner. In `I > y > J`, expression `I` is evaluated. Each value published by `I` initiates a separate and new execution of `J`. Now, the execution of `I` continues in parallel with the executions of `J`. If `I` do not publish even a single value, then there is no execution of `J`.

  - Pruning combinator "«" is a special type of combinator which can be seen as an asynchronous parallel combinator. For example in `I < y < J`, both `I` and `J` execute in parallel. Execution of parts of `I` which do not depend on `y` can proceed, but site calls in `I` for which `y` is a parameter are suspended until `y` has a value. If `J` publishes a value which can be assigned to `y`, then `J`'s execution is terminated and the suspended parts of `I` can then proceed.

    The "»" combinator has the highest precedence, followed by "|" and "«".

- Orc provides several fundamental sites like *Rwait(t)*, *Prompt()*, etc to promote writing efficient programs.

- Orc allows users to define local functions. Function calls act and look like site calls, with a few exceptions:

  - A site call will block if some of its arguments are not available, but a function call does not.
  - A site call can publish at most one value, but a function call can publish more than one value.

  Orc also supports functions, sites as arguments to a function call.

- The recent Orc implementation is allowing Java classes to be used as sites.

2.4.3 Jolie

Jolie (Java Orchestration Language Interpreter Engine) [35] is an orchestration language for services in Java based environment. The statement composers and dynamic fault handling are two important features in this language. In dynamic fault handling [36], instead of statically programming fault handlers, they are installed dynamically at the execution time. This facilitates fine tuning of fault handlers and termination handlers depending upon which part of the code has already been executed.

In Jolie there are basically three statement composers: sequence, parallel and input choice. Statements can be composed sequentially using ";" operator. It means that the statement to the left of the sequence operator is executed first and then the statement to the right of it. The syntax of the sequence statement is as follows.

```
statementx ; statementy
```

`statementx` gets executed first and then the `statementy`. The "|" operator is used to compose statements in parallel. The statements to the left and right of the parallel operator are executed concurrently. The syntax is as follows.

```
statementx | statementy
```

`statementx` and `statementy` are executed concurrently. The third composer, is for guarded input. Here, message receiving is supported for any of the input statements that are listed. When a message for an input statement is received, all the other

branches are deactivated and the corresponding branch behavior is executed. The syntax is as shown in the listing.

```
[IS_1]{branch_code_1}
[IS_2]{branch_code_2}
[IS_3]{branch_code_3}
```

If the message is received on the input statement `IS_2`, then `branch_code_1` and `branch_code_3` are disabled and execution continues through `branch_code_2`. Since CoT is characterized by distributed execution, we believe explicit coordination language support with at least minimal features for coordination and composition, for different work flow patterns is a must for any CoT programming framework.

2.5 Polyglot Programming

Polyglot programming is also called multilingual programming. It is an art of developing simpler solutions by combining the best possible solutions using different programming languages and paradigms. This is based on the observation that there is no single programming paradigm or a programming language which can suit to all the facets of modern day programming or software requirements. It is also called as poly-paradigm programming (PPP)[3], to appreciate the fact that many modern day software combines a subset of imperative, functional, logical, object-oriented, concurrent, parallel, and reactive programming paradigms.

One of the oldest example of polyglot programming is Emacs[4], which is a combination of parts written in C and eLisp (dialect of Lisp). Web applications are generally based on three-tier architecture to promote loose coupling and modularity, it is also a representation of polyglot software systems. Polyglot programming [37] has been observed to have increased programmer productivity and software maintainability in web development.

[3]http://deanwampler.github.io/polyglotprogramming/
[4]http://www.gnu.org/software/emacs/emacs-paper.html

Though the word "Polyglot" was used in software development since 2002, the definition of Polyglot programming is not standardized yet. Many different definitions by polyglot practitioners has been documented in Harmanen [38] and Fjeldberg [37]. One of the definition says, "programming in more than one language within the same context". Another one says, "using multiple programming languages on the same managed run-time". Fjeldberg extends the definition taking into account the developers' perspective as: "programming in more than one language within the same context, where the context is either within one team, or several teams where the integration between the resulting applications require knowledge of the languages involved".

In a Polyglot programming environment, the platform used for the integration and the different programming languages supported by the given platform are the two essential aspects. An inverse pyramid [39] can be used to categorize the programming languages in a polyglot software system. The inverse pyramid has three layers: Stable, dynamic and domain as shown in Figure 2.2

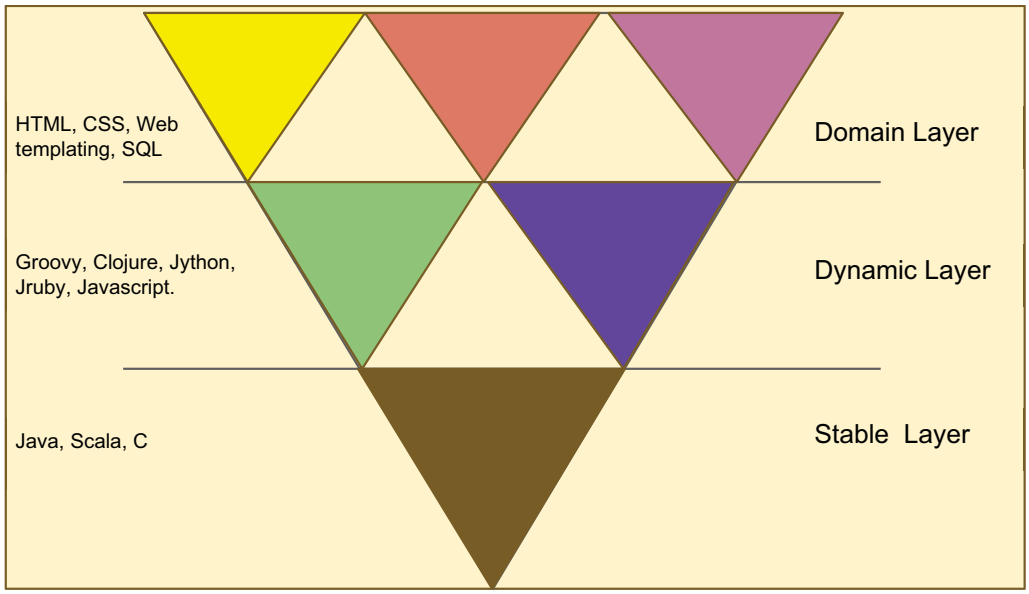


Figure 2.2 – Inverse pyramid for polyglot programming

Statically typed programming languages like Java and C that provide well tested

and stable functionality settle towards the stable layer. The less powerful general-purpose technologies like HTML, CSS which are tightly coupled to a specific part of the application bubble up to the top layer, and the dynamic layer in the middle consist of a variety of programming languages like Groovy, Clojure which are more flexible and aid rapid functionality development. The inverse pyramid signifies the fact that it is the single stable language, which supports all the above layers and various languages in a bedrock fashion.

Since CoT is characterized by heterogeneity in various forms, a single programming language or a single programming model may not be able to provide complete support for the application development in CoT. As we have already argued, at least a coordination language and a computational language is required in a unified programming model for CoT which in a way is polyglot programming.

2.6 Summary

In this chapter we review some of the technologies and programming paradigms from the CoT perspective that serves as a background for our project. Embedded languages review exposes us to the the many existing features for resource constrained programming using C and its flavors. The section on message passing techniques like RPC, REST, CREST and CoAP explains the possible approaches a programmer can take for communication (request/response, publish/subscribe, etc,.) in CoT application development. The polyglot programming section advertises the philosophy of multi-lingual programming and since it has been found advantageous in web application development, we foresee polyglot becoming as a norm in new age application development. Finally, coordination languages push forward the case of having a pure co-ordination language in distributed and parallel programming models. JAMScript is designed as a hybrid coordination language providing Orchestration and Choreography support to processes running JavaScript and C programs.

Chapter 3

JAMScript Design

In this chapter, we discuss the design of JAMScript components, the language and its runtime. The concepts behind coordination languages and the present state of practices in embedded software development have played a huge role in arriving at the present design of JAMScript. JavaScript is the de-facto language in web browser. With Node.js[1], JavaScript brings several advantages like: event handling, scalability, asynchronous, and non-blocking, to Server-side processing, which has catapulted JavaScript to one of the favored language statuses in Server-side programming. The popularity of JavaScript in the web browser, server side and the wide spread use of C in embedded programming inspired us to design an hybrid coordination language, which can combine both of them to provide a single programming framework for CoT.

3.1 Overview of the JAM machine

The JAM machine is a mechanism for organizing the computation that takes across a multitude of computing elements: things and Cloud VMs. The JAM machine uses a tree based architecture. It has two types of nodes: C nodes and J nodes. C nodes are computing devices that run C programs and J nodes run JavaScript programs. The

[1]https://nodejs.org/

C nodes are linked to the J nodes and the J nodes themselves could be connected by root (S) nodes. The S nodes do not have any significance in terms of programability, their purpose is to hold a JAM machine instance together as a connected entity. These S nodes are also called as supervisor nodes.

Figure 3.1 shows a simple JAM machine where multiple C nodes are connected to Cloud-based J node. The JAM machine uses a single program multiple device model which is based on the well known Single Program Multiple Data (SPMD) model for parallel computation. The JAMScript language that is described in the next section, is designed to achieve this goal.

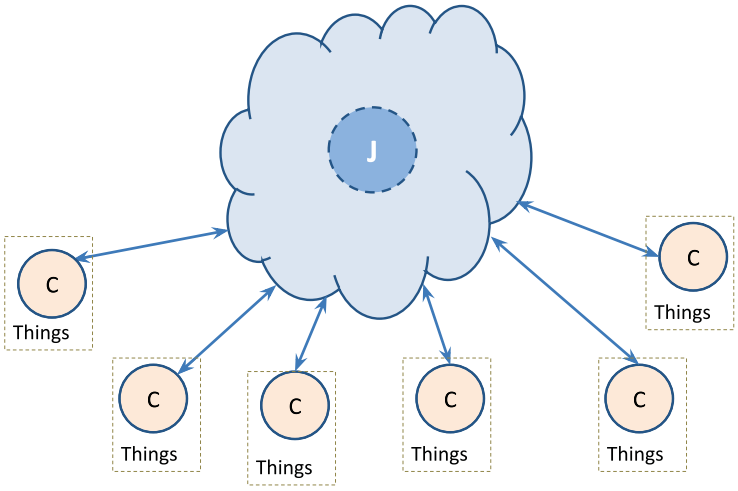


Figure 3.1 – A simple JAM machine

The functions implemented in the C nodes are exposed to the J nodes, so that the J nodes can invoke them to perform the specified processing on them. Similarly, certain functions in the J nodes are exposed to the C nodes as well. Because many C nodes are connected to a J node, the J node has access to the APIs exposed by many C nodes through which the J node could orchestrate the activities of the C nodes.

3.2 JAMScript Language

C programs are a combination of functions. Devices can have many C functions and one of them should be running at any given time with a single thread of execution. A subset of those functions can be exposed for external invocation through the Cloud. Similarly, processing intensive computing can be off-loaded from the things onto the Cloud to exploit the unbounded processing capacities available at the Cloud for computations and data manipulations. Therefore, processing intensive applications can be written in JavaScript and can be hosted on the Cloud, which can be invoked for remote execution by the things. The performance of JavaScript, although significantly poor for general routines, it can be within a factor of 2, of native C implementations, if restricted to subset of the language (e.g., asm.js). JAMScript acts as a glue between the C functions and programs running on the Cloud, providing coordination and control. The remote procedure calls and parameter marshaling that are necessary to perform the distributed activity execution is implemented by the JAMScript runtime. To facilitate coordination and control, JAMScript language provides a new construct called "activity", which defines a new type. An activity is a sequence of blocks that are derived from C functions that are glued together by the constructs introduced by JAMScript. JAMScript allows these activities to be either synchronous or asynchronous. A simple synchronous activity definition using JAMScript constructs is shown in Listing 3.1.

```
1 jamdef [sync] C_func_declaration [in namespace] [requires tag]
2 {
3     //code for the primary block
4 }
```

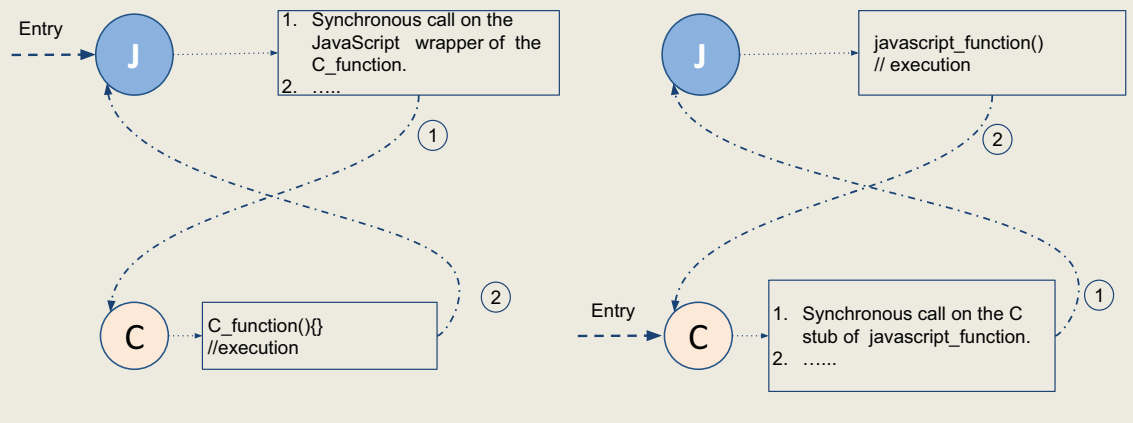
Listing 3.1 – Definition of an activity

jamdef is a keyword of the JAMScript language, which informs the JAMScript compiler that it is an activity definition of a functionality that can be remotely invoked either from the Cloud or from the things. A synchronous activity signified by the inclusion of the `sync` keyword in the *jamdef* declaration is very much like a normal C function – activity (function) call returns on completion with a return

value. Conversely, an asynchronous activity returns immediately with an *handle* to the executing activity. Using the handle, the state of the activity can be checked or controlled – similar to the POSIX threads in C. Further, in the definition syntax, we have the C function declaration which is similar to the practices in C language. The pass-by-value mechanism in JAMScript is little different from the pass-by-value mechanism in C. In C, with pointer arguments a function has the ability to manipulate values in the calling scope. In JAMScript, however, even pointers cannot get back to the values in the outer scope. As a way of design, the JAMScript runtime makes a copy of the invocation parameters which is available to the block of execution. One of the advantages is that the same values will be available even in the case of an activity getting restarted (support for recovery from fault). An optional *in* clause and a name string is used to specify the namespace in which the activity should be posted, which provides scoping containers. The *requires* keyword along with a tag will inform the runtime, the requirements to be meted out for that defined activity to run. The predefined tag can be used to specify requirements like availability of a temperature sensor, a pressure monitor, or some other required sensors and actuators.

The primary segment of an activity can be defined in two different ways: using JavaScript statements for it entirety or C statements for its entirety. If the primary segment is defined using JavaScript, the JAMScript compiler compiles it to a JavaScript component and a C stub. The C stub is generated so that it can be used to invoke the newly created JavaScript function from the C side. On the JavaScript side, this function is posted in a pre-specified namespace location (as specified using the optional *in* clause) and can be accessed by other JavaScript functions too. When the primary segment is defined using C and when compiled through JAMScript compiler, a wrapper function is generated in JavaScript along with the `.c` file of the function. The wrapper can be used to invoke this function and it is made available under the namespace in the Cloud when loaded. The synchronous activity invocation and execution is illustrated in Figure 3.2.

The entry into an activity can happen from both C nodes and J nodes depending upon the type of primary block defined under *activity* definition. If the primary



(a) (b)

Figure 3.2 – Synchronous invocation from J and C sides

block is in C, as required, it will run on a device and will be invoked from the J node through a JavaScript wrapper. Since it is a synchronous activity, it will be a blocking call on the caller. If the primary block is written in JavaScript, then it will get executed on the J node and can be invoked remotely by C nodes. Exceptions are handled by the support provided in the JavaScript language.

Example scenario

A simple example sequence in activity invocation is illustrated here. We want the C function shown in Listing 3.2 to run on devices but called from the Cloud. For this, we need to define it as an activity in the JAMScript environment as in Listing 3.3. In this example, we are defining it as a synchronous activity.

```
1 float cal_interest(float time, float principle)
2 {
3   float rate= 10.0;
4   return (time*principle*rate)/100;
5 }
```

Listing 3.2 – A sample C code running on devices

```
jamdef sync float cal_interest(float time, float principle) in xyz
{   float rate= 10.0;
    return (time*principle*rate)/100;
}
```

Listing 3.3 – Activity definition for the example C function

As shown in Listing 3.3, the primary block specification can have a set of arguments that are passed-by-value to the block at its invocation. When compiled, we will have a `.c` file of the function and a `.js` file for the JavaScript wrapper to invoke this function from the Cloud. The activity in Listing 3.3 can be invoked by a simple JavaScript code running on Cloud (J node) as in Listing 3.4.

```
function total_amount(time, principle)
{
 var interest= xyz.cal_interest(time,principle);
 //JavaScript wrapper function of cal_interest which is running on
   devices. A synchronous call.

 var amount= interest+principle;
 console.log(amount);
}
total_amount(10,100);
```

Listing 3.4 – JavaScript code on J node invoking C function on a C node

Similarly, we invoke functions implemented in the Cloud in JavaScript from C nodes. The following lists the JavaScript function that runs on Cloud that can be remotely invoked from the C nodes on devices.

```
function cal_interest(time,principle)
{
  var rate=10.0;
  return (time*principle*rate)/100;
}
```

Listing 3.5 – JavaScript on J node in Cloud

The above code is defined as an activity using JAMScript constructs as shown in Listing 3.6.

```
jamdef sync float cal_interest(float time, float principle) in xyz
{   var rate=10.0;
    return (time*principle*rate)/100;

}
```

Listing 3.6 – Activity definition of JavaScript function

After compilation, we will have the above function as a JavaScript `.js` file and the C stub to invoke it from the devices as a `.c` file. Now, this activity can be invoked by C nodes through the C stub as illustrated in Listing 3.7.

```
#include<stdio.h>
int main()
{
 float time=5.0;
 float principle=6500;
 float interest, sum;
 interest = xyz_cal_interest(time,principle);
 // a C stub which invokes JavaScript function cal_interest() running
    on a J node. Synchronous call.
 sum= interest+principle;
 printf("The total amount to be paid is: %f\n", sum);
}
```

Listing 3.7 – A C node invoking the C stub of a JavaScript function

Asynchronous activities

The above function invocations can be done asynchronously as well. For that, JAMScript provides a few more additions to the definition of an activity. An activity can have several optional blocks associated with it besides the primary block in the asynchronous mode. The *oncomplete* block is the one that is executed as a callback on the successful execution of the primary block of an activity. If it is a failure, then *onerror* block is executed as a callback. Listing 3.8 shows the format for the oncomplete and onerror block definitons. If the primary block is executed on the device, the callback: *oncomplete* and *onerror* blocks will execute on Cloud and will

be JavaScript blocks. Likewise, if the primary block is executed on the Cloud, then the JAMScript design warrants, *oncomplete* and *onerror* blocks to run on things as a callback. Though the parameters are detached from the ones in the outer scope of the function, they are not immutable. The changes made to the formal parameters during an activity remains available for other blocks until the activity completes.

```
jamdef C_func_declaration [in namespace] [requires tag]
{ // code for the primary block
}
[ oncomplete C_func_declaration()
{// code for the complete block
}]
[onerror C_function_declaration()
{// code for the error block
}]
```

Listing 3.8 – *oncomplete* and *onerror* definition

The interface supported in JAMScript for an activity under asynchronous mode is similar to *Promises* API of JavaScript[2]. An activity call takes two handlers: *completion* and *error*. The completion handler is run when the primary block of the activity finishes its successfully. Similarly, the error handle will be utilised if at all the primary block fails to complete its execution. We refer to the handle returned for an activity call as promises – although JAMScript uses a slightly modified interface.

Example for Asynchronous call

Let Listing 3.9 be a C function available on some device, whose services may be required on the Cloud.

```
float cal_interest(float time, float principle)
{ float rate=10.0;
  return (time*principle*rate)/100;
}
```

Listing 3.9 – A sample C part running on devices

[2]https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise

This code, can now be exposed to Cloud by defining it as an asynchronous activity in the JAMScript. The JAMScript compiler will generate a JavaScript wrapper to be invoked by the J node. Based on the outcome of primary block execution, either *oncomplete* or *onerror* block can be the "callback" function executing on the J node in Cloud. The activity definition with all the three blocks is shown in Listing 3.10.

```
1
2 jamdef sync float cal_interest(float time, float principle) in xyz
3 {   // code for primary block in C, runs on C node.
4     float rate=10.0;
5     return (time*principle*rate)/100;
6 }
7 oncomplete void comp_block(float interest, float principle)
8 {
9     // code for the complete block in JavaScript, runs on J node
10    var amount= interest+principle;
11    console.log(amount);
12 }
13 onerror  void error_block()
14 {
15    // code for the error block in JavaScript, runs on J node.
16    console.log("error in processing");
17 }
```

Listing 3.10 – Activity definition of C function

The return values from the C node is harmonized with the JavaScript program by the *oncomplete* and *onerror* block implementations in JavaScript. This primary block can be invoked by J node as in Listing 3.11

```
1 xyz.cal_interest(10,100);
2 /*JavaScript wrapper function for cal_interest which will run on a C
      node. This call is an asynchronous call. This thread can proceed
      further with other computations, while the events "oncomplete"
     or "onerror" can trigger the JavaScript callback function on
     another thread.*/
3 ....
```

Listing 3.11 – JavaScript wrapper function invocation on J node

The same example can be transformed for the entry from C side. Listing 3.12 is a JavaScript function which can run on the Cloud.

```
function cal_interest(time,principle)
{ var rate=10.0;
  return (time*principle*rate)/100;
}
```

Listing 3.12 – JavaScript on J node in Cloud

The Listing 3.13 does the activity definition with all the three blocks. As, explained earlier, since the primary block is in JavaScript, the call back functions are mandated to be written in C, to facilitate on device execution. The JAMScript compiler generates a C stub on the primary block to be invoked by the C node.

```
jamdef  float cal_interest(float time, float principle) in xyz {
    //primary block code in JavaScript. Runs on a J node.
    var rate=10.0;
    return (time*principle*rate)/100;
}
oncomplete void comp_block(float interest, float principle)
{
// code for the oncomplete block in C. Runs on a device for successful
   callback.
    float amount;
    amount= interest+principle;
    printf("The total amount to be paid is: %f\n", sum);
}
onerror  void error_block()
{
// code for the onerror block in C. Runs on a device for failed-error
   callback.

   printf("error in processing\n");

}
```

Listing 3.13 – Activity definition of JavaScript function

The following Listing 3.14 shows the entry through a C node.

```
#include<stdio.h>
int main()
{
 float time=5.0;
 float principle=6500;
 float interest, sum;
 xyz_cal_interest(time,principle);

 /* a C stub which invokes cal_interest running on a J node. This call
    is in asynchronous mode. This thread can proceed further with other
    computations, while the events ``oncomplete'' or ``onerror'' can
    trigger the callback function in C on another thread. */
 ....
 ....
}
```

Listing 3.14 – A simple Example-on C node

Extensions on Asynchronous Activities

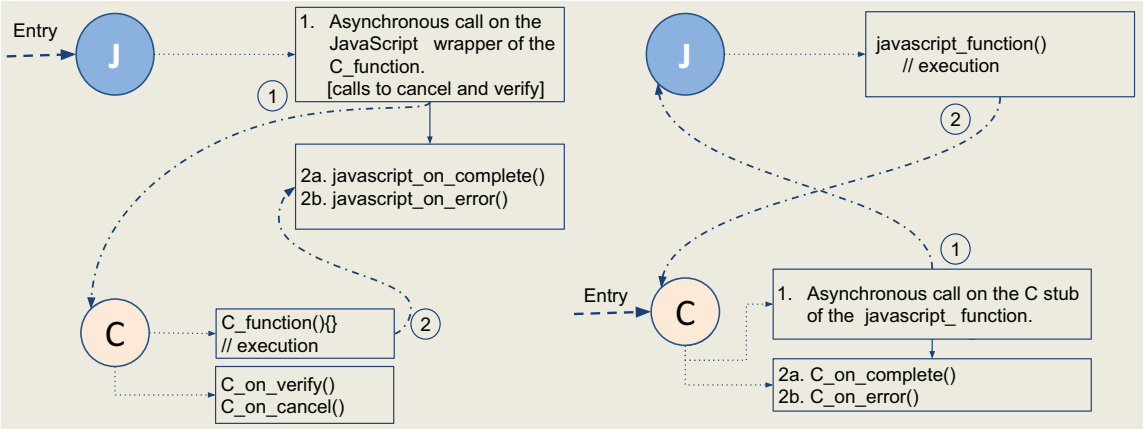
An asynchronous activity can be cancelled by sending a *CANCEL* message to it while it is still running the primary block. This is supported by the runtime. Further, if an activity has already completed its execution, then it can be cancelled only if the activity's cancel block is defined by the programmer. The language runtime has no role to play in this context. The cancel block codifies the sequence of steps to undo the actions of the primary block, as stipulated by the programmer. Similarly, the programmer can use the verify block to define a scheme to independently verify the operation of an activity on the device. Listing 3.15 is the complete format for all the definitions in asynchronous mode.

```
jamdef C_func_declaration [in namespace] [requires tag]
{
    // code for the primary block
}
[ oncomplete C_func_declaration()
{
```

```
7       // code for the complete block. Can be in C or JavaScript.
8   }]
9   [onerror C_function_declaration()
10  {
11      // code for the error block. Can be in C or JavaScript.
12  }]
13
14  [oncancel C_function_declaration()
15  {
16      // code for the cancel block(only in C)
17  }]
18
19  [onverify C_function_declaration()
20  {
21      // code for the verify block(only in C)
22  }]
```

Listing 3.15 – on_complete and on_error definition

Figure 3.3 illustrates the control transfer sequence in asynchronous activities, with entry from both J and C nodes.



(a) (b)

Figure 3.3 – Asynchronous invocation from J and C sides

Table 3.1 summarizes the C and JavaScript combinations for implementing the different blocks of an activity and the entry side.

Entry	Primary	Complete	Error	Cancel	Verify
JS	C (async)	JS	JS	[C]	[C]
JS	C (sync)	-	-	-	-
C	JS (async)	C	C	-	-
C	JS (sync)	-	-	-	-

Table 3.1 – Possible implementations for the activity blocks

Other JAMScript constructs

The activity APIs provided in the JavaScript side is sufficient to invoke the activities and track their execution. However, we need additional support for orchestrating the computing activities taking place within the IoT. These orchestration primitives provided in JAMScript can be used only in the JavaScript side as the J nodes nodes are responsible for controlling the computing activities on C nodes.

The orchestration primitives `seq` and `par` are inspired from the Orc programming language [33] and Jolie [35]. The simplest of the two is the *sequential* composition operator `seq`. Like the activity invocation providing a "promise", regarding the future execution of the activity, this composition operator also provides a promise. The sequential composition operator (`seq`) shown below provides a single promise for the whole sequence.

```
12 seq {activity_1, activity_2, ... activity_n}
```

The "promise" for the whole sequential block will be based upon the promise for the last activity in the sequence. All activities specified in the sequence in left to right order must complete for the promise of a sequence to get fulfilled. When an activity is invoked at a JavaScript node, several C nodes could be eligible to run it. The JAMScript runtime picks a C node at random to run the activity. If it fails, another node which has the capability is picked and so on. The activity fails if no C node

can successfully run it. While the promise of a sequence can get fulfilled in one way, it can fail with n different errors given that there are n activities in a sequence.

Unlike the sequential composition operator that awaits for the completion of an activity before launching the next, the parallel composition operator launches all activities at the same time on capable C nodes. The parallel composition operator (`par`) returns an array of promises – one promise for each activity specified in the parallel composition.

```
13 par {activity_1, activity_2, ... activity_n}
```

The sequence and parallel operators can compose in different ways. A parallel composition can include one or more sequence compositions like the following:

```
14 par {seq {act_a, act_b}, act_x, seq {act_p, act_q}}
```

In this case, the parallel composition returns three promises. Similarly, we can have a sequential composition that include several parallel compositions.

```
15 seq {par {act_a, act_b}, act_x, par {act_p, act_q}}
```

In this case, the sequential composition returns a single promise although it contains parallel compositions inside it. The sequential composition proceeds with the fulfillment of the promise when at least one promise of the parallel composition is fulfilled. All promises of the contained parallel composition need to fail for the sequential composition to fail. In the above example, $\texttt{act}_a$ and $\texttt{act}_b$ are started in parallel and when one of them completes $\texttt{act}_x$ is started. If both the activities $\texttt{act}_a$ and $\texttt{act}_b$ fail in all the eligible C nodes that are capable to run them, then the above sequential composition fails and does not proceed further.

Using the `seq{}` and `par{}` constructs the programmer is able to do sequential and parallel compositions of activities. However, these constructs do not offer mechanisms to place an activity on nodes with specific attributes. We provide operators based on MapReduce programming model [40]. The runtime chooses the C nodes that have capability in random to run the activities. A particular activity can be mapped by the programmer onto all nodes with given attributes by using the *map* construct.

```
16 map {activity_x@predicate_y}
```

At least one promise is returned by this construct. When the predicate specified for the `map{}` does not match any C node, the promise fails with an error. Otherwise, the construct returns a promise for each node onto which the activity is mapped. The *map* operator can be composed as part of *seq* and *par* compositions.

```
17 par {act_b, map {activity_x@predicate_y}, seq {act_p, act_q}}
```

As earlier, this composition will return an array of three promises, including for the `activity`$_x$`@predicate`$_y$. Similarly under sequential compostion.

```
18 seq {map{act_a@predicate_b}, act_x, par {act_p, act_q}}
```

Here, if the *map* fails, then the whole sequence fails.

The final composition operator provided by JAMScript is the aggregator. It takes an array of promises and applies a user-defined aggregation function.

```
19 red {promises_x, func_y}
```

A single promise is returned by this construct. The `func`$_y$ supplied by the programmer will determine when the promise will be fulfilled and under what conditions. When the promise is fulfilled, all outstanding activities are cancelled by the `red{}`. The activity construct provides comprehensive support for orchestrating computing activities within CoT. For example, it allows the computing tasks to be sequenced in arbitrary ways and placed on devices with given attributes or the Cloud.

We expect CoT to handle large volumes of data particularly in configurations that involve sensors. Therefore, JAMScript proposes a data-driven coordination mechanism to handle data flows from the devices to the Cloud. One of the unique aspects of our data-driven coordination mechanism is its ability to pre-structure the data injected by the devices according to the data definitions created by the programmer.

To push data from the C node to the Cloud, we use the idea of *live* variables [7], where `live` is a new storage class introduced by JAMScript into C. Any updates to a live variable immediately gets propagated to the Cloud. A C code fragment with a live variable is shown below.

```
20 live double x;
21 ...
```

```
while(...) {
    x = func(..);
}
```

CoT can generate data that is time dependent (organized as time series) or event dependent. Event dependent data could be standalone values that can have meta data associated with them to explain the event that generated them. The purpose of data pre-structuring is to improve the efficiency of data handling such that data generated by the devices can be processed by the Cloud resident programs with minimal latencies, minimal storage overheads, and minimal programming steps. An example data definition in JAMScript is shown in Listing 3.16.

```
jamdata {
    type_x = jtypedef {attr_1, attr_2, ...};
    type_q = jtypedef {attr_a, attr_b, ...};
    x = timeseries(100) of type_x;
    z = timeseries(200);
    q = simple of type_q;
    y = array of x;
    p = olist of q by attr_a;
}
```

Listing 3.16 – An example JAMScript data definition

All the live variables are mapped by JAMScript into a global scope. So no two live variables can have the same name. Listing 3.16 shows that live variable `x` is a time series with up to 100 elements. The data definition also shows variable `y` as an array of `x`. Suppose several C nodes are generating values for `x`, these values can be accessed through the array `y`. The cardinality of `y` indicates the maximum number of C nodes pushing data into the Cloud.

3.3 Support for fault tolerance

JAMScript relies on a new computing model where all computing elements in the CoT run portions of the same program. A JAMScript source can have C segments,

activity definitions, and JavaScript segments. Once the source is translated by the JAMScript compiler we get C and JavaScript functions. Some activities can be disabled on certain nodes if the nodes do not meet the requirements specified by the programmer (see Listing 3.15) in the activity definition.

A single JAM machine is more like a computing cluster, with computing dispersed amongst J and C nodes. We propose to have centralized components like management/supervisor nodes for node monitoring responsibilities with heartbeat system [41] on cloud. In addition to the J and C nodes, the runtime introduces S (supervisor) nodes for fault tolerance purposes. These supervisor nodes listen for periodic messages (heartbeat) from all the J and C nodes of a single JAM machine. The S nodes are part of the JAM machine model (as shown in Figure 3.4) but are transparent to the programmers.

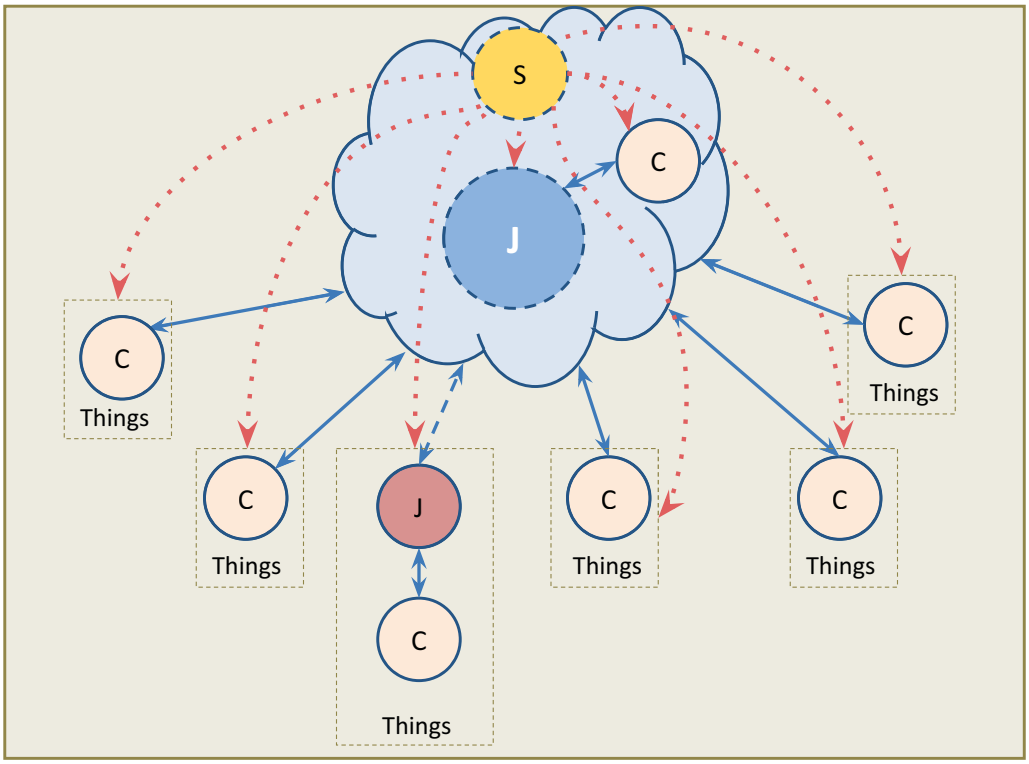


Figure 3.4 – Complete JAM machine configuration

Normally, devices run C nodes and Cloud runs the J nodes. And both nodes are connected to the S nodes so that any failures of C or J nodes can be detected by the S nodes. The configuration of a JAM machine can be reorganized by the S nodes depending on the loading conditions. A device could run a J node in addition to a C node, so that the device could have full JAMScript application running while it is disconnected from the Cloud. Similarly, the Cloud could run C nodes to perform compute intensive tasks in the Cloud. Also, to support applications which are latency sensitive, the J nodes can be placed in the infrastructure nodes that are close to the devices thereby minimizing the latencies while maintaining high reliability.

Due to their importance and small fraction they compose, we propose to extend fault tolerance for S nodes. The most common means of providing fault tolerance for centralised components is to provide software based "active replication" [42]. As in active replication, at any given moment, there will be two Supervisor (S) nodes, monitoring all the J and C nodes in an instance of a JAM machine. Also, these S nodes keep track of each other, and whenever S1 or S2 fails, the supervisor node alive instructs the runtime to immediately create and initialise another supervisor node to replace and initialise the failed one.

3.4 Programming patterns

Using JAMScript, we believe, many programming patterns can be easily implemented. Below, we provide a few example patterns.

1. *Spawn a task on many computing elements.* With many sensor-based IoT applications it is necessary to spawn a task such as "take a sensor reading" across many sensors. With JAMScript, we can use the `map` construct to run a given activity over a defined collection of sensors. And we can use `red` to obtain a reading based on a programmer defined function.

2. *Chain of tasks across different computing elements.* Using the `seq` construct we can run activities such that one is dependent on the other. Failure of one

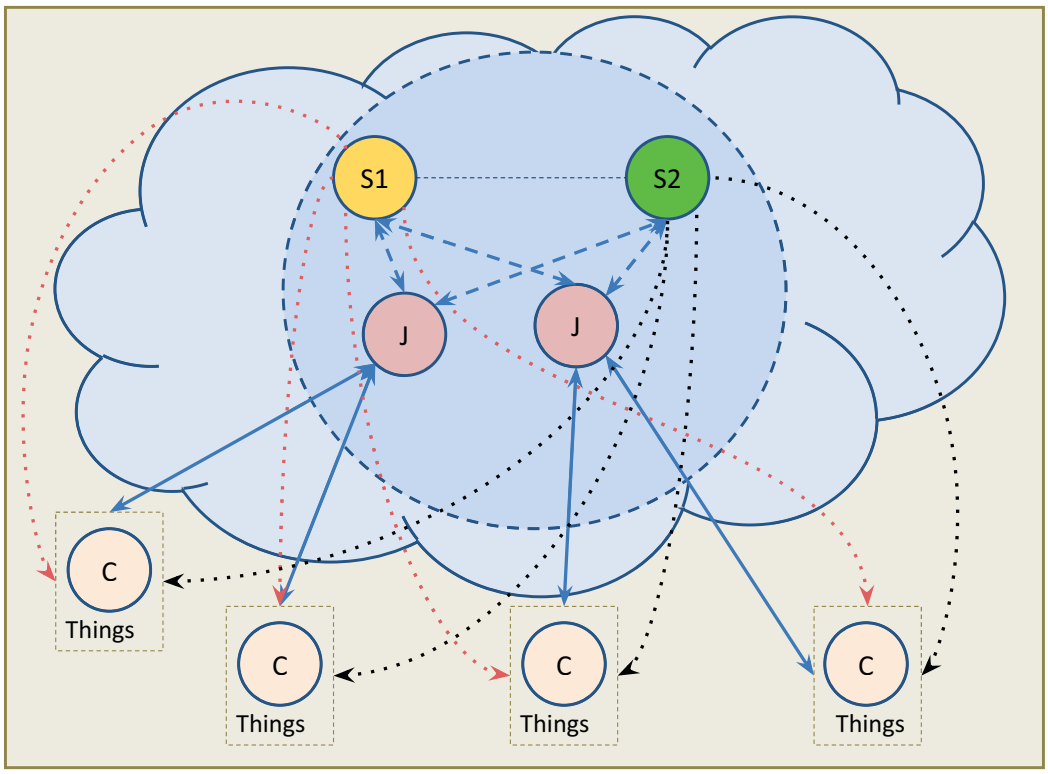


Figure 3.5 – Fault tolerance support on Cloud

activity will terminate the execution of the chain of tasks with the appropriate error condition.

3. *Chain of tasks with fault tolerance.* Even the `seq` construct has fault tolerance built into its execution because it tries the different C nodes that could run an activity when it receives a failure.

4. *Offloading computing from Cloud to native execution.* By default the J nodes execute in the Cloud. When we have time consuming computing tasks, it is necessary to execute them natively. Using JAMScript we can place the time consuming task in an activity running in a C node. However, instead of mapping the C node to a device, we can place the C node in the Cloud itself.

5. *Offloading computing to Cloud.* Devices can conveniently offload computing to the Cloud by calling an activity whose primary block will run on the Cloud. The Cloud could in turn offload the specified computing task to yet another location – for example, for native execution as per above pattern.

Chapter 4

Implementation

In the previous chapter, we discussed the design of JAMScript programming framework. In this chapter we present the implementation details of the language compiler and the run time. At this time, a prototype of the JAMScript compiler as described here is working. However, work is still ongoing to improve the language and compiler[1].

4.1 JAMScript compiler

The JAMScript compiler compiles valid JAMScript programs in files with `.jm` extensions to to C and JavaScript files. An activity definition in JAMScript is in C like syntax, as follows.

```
1 jamdef [sync] C_func_declaration [in namespace] [requires tag]
2 {
3     // code for the primary block in C or JavaScript
4 }
5 [ oncomplete C_func_declaration()
6 {
7     // code for the complete block in C or JavaScript.
8 }]
```

[1] This work is carried out in collaboration with Robert Wenger and Professor Maheswaran. My role is in documenting and testing to facilitate wider participation in the project.

```
[ onerror C_function_declaration()
{
    // code for the error block in C or JavaScript.
}]

[ oncancel C_function_declaration()
{
    // code for the cancel block in C
}]
[ onverify  C_function_declaration()
{
    // code for the verify block only in C
}]
```

Listing 4.1 – JAMScript activity definition

As we have already discussed, the primary block can be in either JavaScript or in C. If the primary block is in C, when compiled, the JAMScript compiler will generate a `.c` file containing the primary C function and a JavaScript wrapper in `.js` file for that C function, which can be invoked by the J node in the cloud under the given namespace. Similarly, if the primary block is in JavaScript, then the JAMScript compiler will generate `.js` file with primary JavaScript function and the respective C stub in `.c` file for invoking the primary block from the C nodes. Further, for asynchronous activity definitions, the compiler will generate callback functions for *oncomplete*, *onerror*, *onverify* and *oncancel* events. If the primary block is in C, then the callback functions for *oncomplete* and *onerror* events will be generated as JavaScript components. For *onverify* and *oncancel* events, if the programmer has defined the blocks, then the compiler will generate equivalent C components. Likewise, if the primary block is in JavaScript, then the compiler will generate the callback functions in `.c` file to run on the C nodes for *oncomplete* and *onerror* events. The output of JAMScript compiler is a bundle of `.c` and `.js` components. Figure 4.1 illustrates the complete compilation process of a typical JAMScript program. The C files are then compiled for respective hardware using the native C language compilers.

The `.o` files and `.js` files are stored in an archive file with a manifest in TOML[2] format. The archive is considered the JAM executable with a `.jxe` extension.

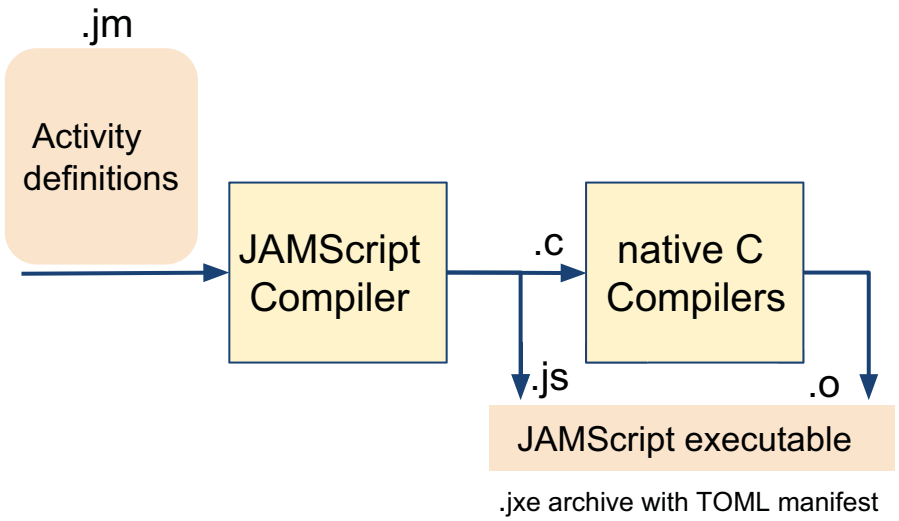


Figure 4.1 – JAMScript program compilation

JAMScript keywords

The JAMScript language extends the C language with new keywords as in Listing 4.1; their individual responsibility has been explained in chapter 3.

```
1 jamdef, jamdata, sync, in, oncomplete, onerror, oncancel, onverify,
2 live, requires, seq, par, map, red.
```

To compile the new language constructs, JAMScript uses OMeta[3], which is a general purpose pattern matching language based on Parsing Expression Grammars (PEG) [43]. Using OMeta, the parser and translator which extends C language is built. The dependency between C language, OMeta and JAMScript is illustrated in the following listings.

The general steps followed to build language translators using OMeta is to:

[2]https://github.com/toml-lang/toml
[3]http://tinlizzie.org/ometa/

- represent the grammar of a language in OMeta language;
- compiling the grammar using OMeta compiler;
- using the output object to match and translate input streams.

The standard C grammar is represented in OMeta language (OMeta/JS) and compiled through the OMeta compiler to generate a C language parser which can parse and generate ASTs for a valid C construct.

```
ometa CParser  {// C language grammar in OMeta language format
}
```

Because, JAMScript extends the C language, the C parser generated using OMeta can be inherited by the JAMScript C parser to build a parser for the JAMScript constructs.

```
// require ES5 parser.
ometa JAMCParser <: CParser {// JAMScript grammar in OMeta language
}
```

Also, since the JAMScript constructs can have JavaScript components in the primary and callback blocks, the `JAMCParser` utilises the service of a JavaScript parser: ES5 parser, written in OMeta provided under ES5 package. Thus, the JAMScript parser can parse and generate Abstract Syntax Trees (ASTs) which can be translated to respective C or JavaScript components. JsonML[4] is used to represent the abstract syntax tree internally. The JsonML package provides the functionality that is required to work with ASTs based on the JsonML data structure. The "factory" method in the package is used to create the node-constructors for each node type that can be used to build the AST. The translators for translating ASTs to their respective code are also built using OMeta with the help of JsonML package. The package also provides a walker implementation that is written in OMeta/JS to traverse through the ASTs built in the parsing phase.

```
ometa CTranslator <: JsonMLWalker {// translation listings for C
   language constructs
```

[4]http://www.jsonml.org/

```
}
```

First, the C translator is built and then the JAMScript translator is built by inheriting this C translator along with ES5 translator (for JavaScript components).

```
//require ES5 translator.
ometa JAMCTranslator <: CTranslator {// translation listings for
    JAMScript constructs
}
```

Figure 4.2 illustrates organization of the JAMScript compiler.

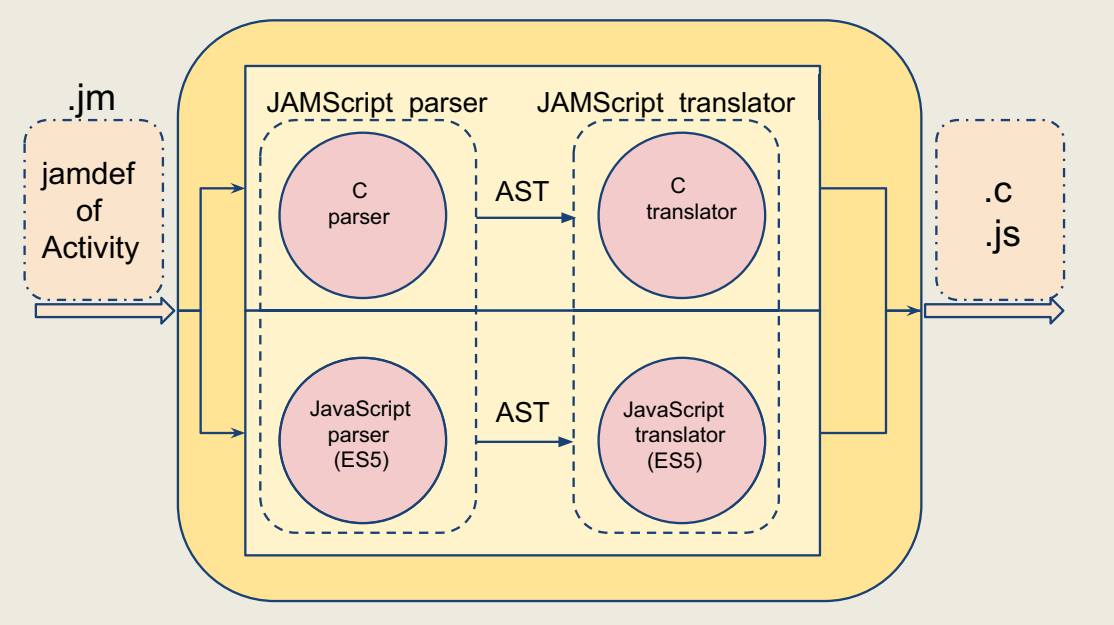


Figure 4.2 – JAMScript Compiler components

JAMScript executable

The `.c` and `.js` files will be the output of the JAMScript compiler, for activity definitions in JAMScript. Further, the C object files can be generated using native C language compilers. The C object files and corresponding JavaScript components representing activities for a particular application are packaged together into

a single archive called JAMScript executable file. The organisation of a JAMScript executable file `.jxe` is described in a TOML formatted manifest file included in the archive.

```
Type = JXE
Description = "JAMScript Executable File"
Name = "xxxx"            # Application name
Version = 1.0            # defines the version of executable  format
   .
List-of-Activities=[...] # name of all activities

[js]
Functions = [array of entry points]
Code = filename in archive.
-----------------------------------------------------------------
[c]
[c.main]                 # main function
Release = X              # a release number
[x86]
Code = filename in archive.
Checksum = "xx"
[arm64]
Code = filename in archive.
Checksum = "xx"
-----------------------------------------------------------------
[c.act_name1]            # activity 1
Release = X
Requirement = [array of attributes required]
[x86]
Code = filename in archive.
Checksum = "xx"
[arm64]
Code = filename in archive.
Checksum = "xx"
-----------------------------------------------------------------
[c.act_name2]            # activity 2
Release = X
```

```
Requirement = []
[x86]
Code = filename in archive.
Checksum = "xx"
[arm64]
Code = filename in archive.
Checksum = "xx"
```

Listing 4.2 – Organisation of an example JAMScript executable

The release number in the executable file can be used by the loader to match, while loading new components to avoid loading same versions of the software module more than once. If the loader determines that a new version is available, then it is loaded and the application can be restarted.

4.2 JAMScript runtime

The JAM library and loader are also part of the runtime of the language. To support transactions between the cloud and the things a set of APIs are provided as part of the JAM library. The TOML formatted manifest in the .jxe file will have all the information about the application, such as the following:

- App name
- Activity list
- All the necessary files (`.o`, `.js`) to load

The JAMScript loader loads the executable to the C node. The syntax is as follows.

```
1 jamload myprog.jxe tag
```

To initiate interaction between things and devices, a connection should be initiated to the server using `jam_init()` API at the C node. It takes two arguments: name of the server and port number. If the server is not down, then it initialises the connection between C node and the server in the cloud (J node). Also, the JAM library APIs are initialised for the C node.

```
int init_jam(char *jam_server, int port);
```

Now, the C node can request the J node (server) to open an application it intends to work with, by giving the application name to `open_application()` API.

```
Application *open_application(char *appname);
```

The server will query its internal table for the status of the application requested by the C node. There are three possibilities: the application may be running; present on cloud but not running; and third – it may not be available in the server.

1. If the application is already running, then the C node will get the list of all activities in that application from the server. The C node can then delete those activities for which it does not have capability. To facilitate this matching, we stress the use of `requires` in activity definitions. The C node maintains this supported activity list for the application in its local table and also forwards it to the server. The server will maintain this capability list of the C nodes to facilitate `seq{}` and `par{}` compositions. The server will connect the C node to the servlet running the application.

2. If the application is not yet created or for the remaining possibility that it may not be running (we will assume the server has a older version) then the C node can get the server to register the application. The C node will send the JavaScript components for callbacks to the server along with the list of capable activities on the C node. The server will store the new application along with meta information. Further, it creates a servlet for the application and connects the C node to the servlet.

If the server realises that the thing has a newer version of the application during the initial handshake of `open_app()`, then the newer version of the app gets registered and the application will be restarted. The invocation of `init_app()` at the C node will install the application from the `.jxe` file on the local node based upon the capability table and starts a servlet for the application and gets the handle to the C runtime. Also, callbacks are registered and the background event loop is started.

One of the problem of incremental loading is, when a call back function is registered but the function is yet to be loaded on the C node and there is a remote invocation on that function. In these sort of scenarios, till the loader loads and updates the server about the status, the registered call backs can be pointing to a dummy function which returns an error status.

The events can be `oncomplete`, `onerror`, `onverify`, `oncancel`, and `oncallback`. The API for registering callback has the following syntax.

```
3 void register_callback(Application *app, char *aname, EventType etype,
    EventCallback cb, void *data);
```

It takes in application name, activity name, event type and the supporting function and data. An application can be closed by using the `close_application()` API, which informs the server to close that particular application. The server will kill the servlet for that particular application. The syntax is as shown.

```
4 int close_application(Application *app);
```

To un-install an application the `remove_application()` API can be used whose syntax is as follows. This removes the application from the server's internal list.

```
5 int remove_application(Application *app);
```

The API `print_application()`, will print the application details: Name, appid, state, server, and port number. It can be used as follows.

```
6 void print_application(Application *app);
```

Remote execution of services can be requested by the C node, through the `execute_remote_func()` API. The syntax is as shown. The application name, the function name (activity in JavaScript) and the input parameters for the function are the input for this API.

```
7 int execute_remote_func(Application *app, const char *fname, const char
    *fmt, ...);
```

To raise an event exclusively, the following API may be used at the C node.

```
8 int raise_event(Application *app, char *tag, EventType etype, char *
    cback, char *fmt, ...);
```

As of now, the compiler for the C part of JAMScript, the above APIs, and a TOML file parser to aid in loading has been implemented. In the next phase, the JavaScript part of the JAMScript compiler including support for composing operators and fault tolerance mechanism from runtime will be implemented.

Chapter 5

Potential Application Scenarios

5.1 Smart roads

One of the important aspect of smart cities are smart roads – within the city and which connect the cities. Smart roads can make it easy for commuters, vehicular traffic planners and managers. Features like traffic density alerts, CO_2 level monitoring at traffic junctions, interactive lighting, etc add on to the features of a smart road.

Interactive Lighting

For smart roads, an important attribute is interactive lighting. By using motion-sensor lights, interactive lights can save lot of energy and at the same time provide better services to highway commuters. When a vehicle approaches a particular stretch of the highway, the motion sensors can actuate the light-on for that section of the road. It will become brighter as the car moves closer to the pole and will slowly dim away as it passes.

For those long stretching highways, which may be less travelled during late hours, interactive lighting can provide optimal night visibility and at the same time can cut costs. JAMScript can be used to develop software for these systems. The C nodes located on the poles can run a program for sensing vehicle movements and actuate

the light accordingly. The central J node can collate the data and can help to make policy decisions for the highway spans and implement them.

5.2 Smart classroom

One of the attribute of a smart building is temperature monitoring. In a smart classroom each instructor will be given services based upon individual requirements like light settings, audio level, temperature level, etc,. Individual preferences can be serviced by things without any manual intervention after the initial settings. Here, we illustrate the scenario of room temperature monitoring using JAMScript. All the things will house C nodes, while the J (server) node will be in the Cloud. For temperature monitoring we assume three devices: a temperature sensor, an heater and a cooler to be present with temperature sensing capability in a class room. Let the application be called as "smart_temp".

1. Let the temperature sensor abstracted as a C node, be the first to get connected to the JAM server. As we saw in the previous chapter, through runtime APIs, the C node can push all the JavaScript components to the server and start the background event loop. The server will include only those activity which is supported by this particular node in the application list. Let the callback component on Cloud after sensing the temperature be `js_action1()` which receives the temperature value from the sensors on C node. Based on the temperature value, it can invoke a method to decrease or increase the temperature on the cooler and the heater.

2. In the next step, the heater gets connected to the server and send its node information to the server along with request for opening of the app "smart_temp". The server collects the information from this node and activates the activities supported by this node. This node can increase the room temperature. The callback function for this node shall be `js_action2()`, which can invoke a request on the temperature sensor to sense the temperature after increasing the temperature.

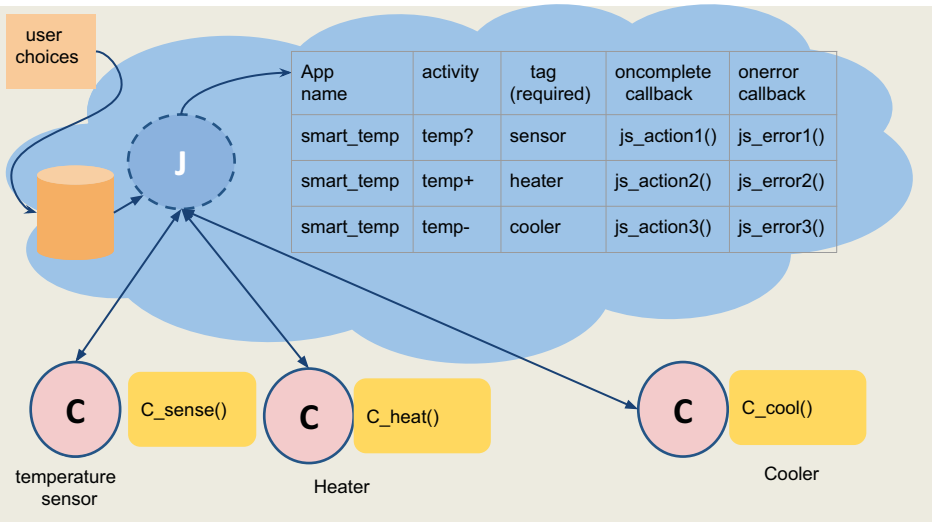
3. Finally, the cooler gets connected to the server and send its node information to the server along with request for opening of the app "smart_temp". The server collects the information from this node and activates the activities supported by this node. This device can decrease the temperature. The callback function `js_action3()` can invoke request on the temperature sensor to sense the temperature after decreasing.

In Figure 5.1(a), there are three C nodes, getting connected to the J node and exporting information about their capabilities. The first C node has functionality to sense the temperature, the second C node can increase the temperature, while the third C node can decrease the temperature. The respective callbacks for successful completion and failure are exported to the J node. While, in Figure 5.1(b), a C node, maintains information about the node's capabilities.

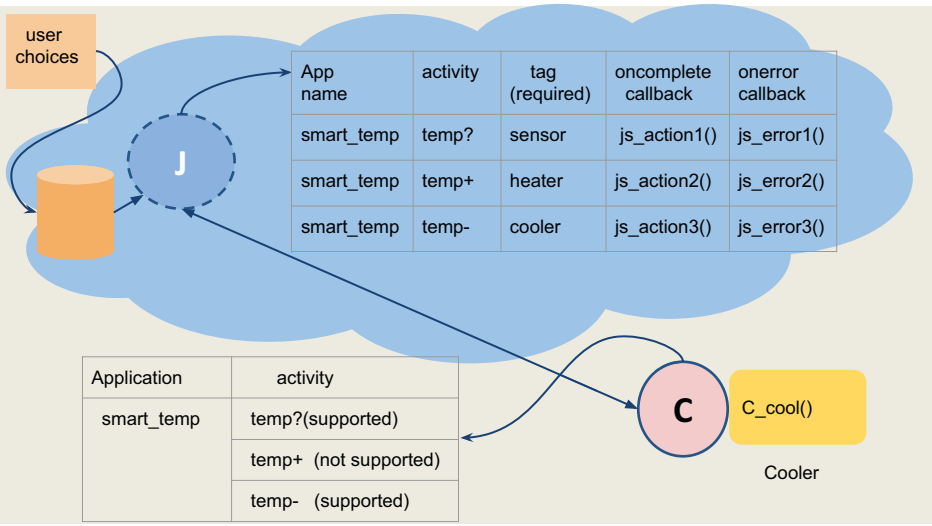
5.3 Health monitoring for elderly

CoT can significantly improve the quality of life for the ever increasing number of elderly people. As an example, a small, wearable device can detect a person's vital signs like heart rate, body temperature and send an alert to health-care professionals whenever a certain threshold has been reached. Also sensors can detect if at all a person has fallen down and send alerts to the emergency services.

In the above scenario, JAMScript can be used to implement the managing software. The sensor devices can have C nodes pushing data through `live` variables to the J node, which can be on a local server. Further, even when there is no Internet connectivity, the J nodes can be on the devices itself and through other alerting techniques like SMS messaging, can inform the emergency services (like 911). Once the connectivity is established, the J node can heal itself to the Cloud.



(a)



(b)

Figure 5.1 – Deployment scenario

Chapter 6

Related Work

6.1 Overview

Programming frameworks typically promote design reuse, implementation reuse, and validation reuse, thereby enhancing software extensibility, flexibility and portability. The complexity of the domain and maturity of the problem are the biggest challenges in developing frameworks [44]. Since CoT domain itself is in initial stages, many frameworks too are in the development and experimental stages. In this chapter, we present – the key support features a programming framework should provide for CoT programming in section 6.2, development methodologies currently being used in application development, for constrained environments in section 6.3, and section 6.4 is a brief description of different programming frameworks for IoT-Cloud that have been recently developed.

6.2 Essential features of CoT Programming Frameworks

We propose the following set of minimum features to be fulfilled by the programming frameworks for CoT.

1. *Coordination*: A CoT can have computing elements playing different roles: controllers, storage managers and application processors. We need programming language support for orchestrating their activities. The orchestration can be explicit (control driven) or implicit (data driven).

2. *Heterogeneity*: Disparate computing devices are brought together by the CoT for the purposes of running smart computing applications. The programming framework should be capable of efficiently exploiting the system heterogeneity by allowing the developer to provide guidance on how the computations must be mapped to the computing elements.

3. *Scalability*: For CoT to be successful, it is not sufficient to just interconnect massive number of devices. They should be programmed to run many creative applications, such that large number of users would benefit from their deployment. Therefore, CoT needs programming frameworks that support variety of programming patterns and also should be able to perform load balancing dynamically.

4. *Fault tolerance*: In CoT, we can expect frequent system partitioning due to mobility of computing elements. The programming framework should allow developers to create applications that can gracefully go between online and offline states as networks partition and heal their connections.

5. *Lightweight footprint*: The programming framework should be lightweight both in terms of the runtime overheads and the programming effort needed by the developers.

6. *Support for latency sensitive applications*: CoT will have many applications which would be geographically distributed and may be latency sensitive. Pushing all the computations to Cloud will not help these sort of applications. The programming framework including the runtime has to support this sort of requirements dynamically..

6.3 Programming Approaches for Constrained Environments

The following four approaches are used predominantly in IoT-Cloud application development [45].

1. *Node Centric Programming.* Here, every aspect of application development, communication between nodes, collection and analysis of sensor data, issuing of commands to actuator nodes has to be programmed by the application developer. Though, there is better control on the way programs work, it is too labor intensive and does not promote portability.

2. *Database Approach.* In this model, every node is considered as a part of a virtual database. Queries as part of an application can be issued on sensor nodes by the developer. This model does not support application logic at this level, rendering it to be of little use in IoT application development.

3. *Macro Programming.* In this methodology, application logic can be specified and also abstractions are provided to specify high-level communication, thereby hiding low level details from developers aiding in modular and rapid development of applications.

4. *Model Driven Development.* It takes note of both vertical and horizontal separation of concerns. Vertical separation increases level of abstraction, thereby reducing application development complexity. Horizontal separation of concern reduces development complexity by describing the system, using different system views. Each perspective elaborates certain aspect of a system.

Many of the IoT/CoT development kits, which are available in the market support one of the approaches listed above. This categorization is not exhaustive; as new hybrid approaches may evolve as the IoT-Cloud domain itself matures.

6.4 Existing IoT-Cloud Frameworks

The IoT/CoT research communities from many academic and research organizations are constantly striving to simplify the efforts involved in application development by developing new programming frameworks. We present a few of them and highlight their key features.

6.4.1 Mobile Fog

Cisco has proposed a new computing model called Fog computing [46]. Here, generic application logic is executed on resources throughout the network, including routers and dedicated computing nodes. In contrast to the pure Cloud paradigm, fog computing resources perform low latency processing near the edge while latency-tolerant, large-scope aggregations are performed on powerful resources in the core of network (Cloud).

Mobile fog [47], extends fog computing by providing Platform as a Service (PaaS) programming model for IoT application development to simplify the task of application development that runs on heterogeneous devices distributed over a wide area and also to provide support for dynamic scaling based on their workload.

Here, an application will contain processes distributed throughout the fog computing infrastructure, Cloud and on edge devices based on geographical proximity and hierarchy. Each process can perform tasks with respect to its location and level in the network hierarchy like sensing, actuation, and aggregation. A process running on a device which is at the edge is a leaf node while a process in the Cloud is the root node in a given hierarchy. Processes on nodes between devices and Cloud are intermediate nodes (routers, servers, etc). Each process handles workload from a certain geo-spatial region.

Mobile Fog provides API support through its runtime. Mobile Fog uses computing instances requirement to provide dynamic scaling. It is based on user-provided policy such as CPU utilization rate, bandwidth, etc.

6.4.2 ELIoT (Erlang Language for IoT)

Though the language Erlang was originally designed for embedded platforms, over a period of time it amassed a complex infrastructure, which is usually not required in devices and is a burden on resources constrained things. ELIoT [48], Erlang language for IoT, tries to address this for IoT application development.

ELIoT provides a small library for developing decentralized sensing/actuation systems, an interpreter suited for resource constrained IoT devices and a simulator for testing the implementations in a fully or partially simulated environment. The ELIoT's virtual machine is a stripped down, lightweight version of Erlang's virtual machine. Heavy libraries, which are not required for IoT are removed (like CORBA middleware systems). It includes a custom-networking stack for improving efficiency and for supporting new communication primitives. Instead of TCP, UDP is used for both reliable and non-reliable communication. A customized reliability layer is built on UDP.

Generally for IoT applications, strict layering of the networking stack may not be fully advantageous; some form of cross layering is found to be helpful for IoT applications [49]. Erlang's network driver fills up the incoming message queue of the receiver only with the payload of the message, hiding all the other details; whereas, the network driver of the ELIoT exposes additional information like the IP address of the source node and the Received Signal Strength Indicator (RSSI) coming from the radio, which are treated as any other type of data.

The Erlang's uni-cast interprocess communication operator ! is built on a complete TCP/IP stack, ensuring reliable communication for both local and remote communication. Since, TCP/IP stack comes with a cost and can be resource draining on devices, in ELIoT, the ! operator is used only for communication between local processes. Remote communications in ELIoT is handled by a set of specific functions

from the ELIoT library, whose semantics is best effort and, is limited to single hop in wireless networks. Further, the ELIoT library supports rich set of communication patterns including the broadcast mode.

ELIoT provides a simulator for supporting IoT application debugging and testing. The simulator can model a complete system through virtual nodes running unmodified ELIoT code. Also, it can run a mixed deployment where virtual nodes seamlessly interact with physical devices. The ELIoT simulator allows debugging a system in a fully simulated deployment environment, which, seamlessly can move into actual deployment environment. The ELIoT framework provide wrappers on nodes which are basically RESTful interfaces, through which nodes can be accessed by users through the normal HTTP operations.

ELIoT brings in the advantages of Erlang to IoT in a light weight framework.

6.4.3 Compose API

Compose API [50] is an IoT service provider platform through RESTful APIs, wherein things, users, and the Compose platform can interact with each other to provide services based on IoT called Internet of Services (IoS). Compose platform is based on Web of Things (WoT): all the physical objects connected to the platform are web enabled and can interact among them using the web protocols. Along with the APIs, the Compose platform consists of GUI, semantic registry, cloud runtime and communication libraries.

Any object which implements the communication protocols of the Compose API is web enabled and is called a Web Object (WO). Each WO holds a virtual identity inside the Compose platform called the Service Object (SO). The SOs communicate with the external WOs through APIs. SOs can act as data endpoints or they can also act as

intermediaries, feeding processed data to other SOs. Every time a sensor attached to WOs produces a new reading, it is forwarded as a Sensor Update (SU) on a stream to the Compose platform to be collected by the corresponding SO for processing based on some processing logic. The processing logic is a combination of logical, string and arithmetic operations implemented in the form of a processing pipeline. A SU goes through a number of stages in the pipeline in order to transform into a new output or a new SU. Connections between SOs are built through subscriptions and

communication between them are through events. A JSON document description deploys each SO in the platform.

Compose API simplifies node centric programming and exposes nodes through RESTful APIs which can be further composed. Such a programming methodology is quite advantageous to IoT.

6.4.4 Distributed Data flow support for IoT

In this approach [51], existing IoT data-flow platforms like WOTkit[1] processor and Node-RED[2] are extended to support distributed data flow, which is one of the important characteristic features of IoT. Data flow programs are generally called flows consisting of nodes connected by "wires". The data-flows are generated using JSON documents. During execution, nodes get instantiated in the memory and the code is executed as and when the node receives data on the incoming "wire".

The nodes do not share states with each other and are inherently independent and can execute code in parallel. This facilitates computation migration between heavy processors and devices seamlessly. Based on user choices and trade-offs computations can be split and distributed, so that a part of it can execute in the cloud while the other parts can execute on edge devices. According to the authors, the present day IoT data-

flow platforms needs to be extended to support distributed data flow for which three things are necessary: flow ownership, naming of nodes, and classification of connections (wires) as local or remote. This framework aims to incorporate the above three attributes to WOTkit and Node-RED to aid in IoT application development from the data flow perspective.

6.4.5 PyoT

PyoT [52], is a programming framework for WSNs, which have the capability to communicate with each other through the Internet using 6LoWPAN and CoAP.

[1] https://wotkit.sensetecnic.com/wotkit/
[2] http://nodered.org/

PyoT abstracts WSNs as software objects, which can be manipulated and composed to perform complex tasks. PyoT uses CoAP's RESTful interface to interact with nodes. Applications can consider sensing and actuating capabilities of nodes, shared with the external world through URIs. The users can discover available resources, monitor sensors and actuators, store data, define events and actions, and program to interact with resources using Python. PyoT supports "in-network processing", in which a part of the application logic can be directly run on devices.

PyoT has five components: i) Virtual Control Room, ii) Shell, iii) Storage Element, iv) Message queue, and v) One or more PyoT Worker Nodes. The Web user interface is the virtual control room that allows execution of basic operations like listing of resources, sensor monitoring, data storage. The Shell allows macro programming for defining complex operations through a set of Python APIs for interacting with resources, which are abstracted as Python objects. The Storage Element maintains the system status. Each PyoT Worker Node generally manages an IoT based WSN, by providing a set of processes that perform generic tasks and support communication activities with other

nodes. The PyoT Worker Node also keeps track of nodes and their resources, provides update to Storage Element, performs sensor data collection and also supports event detection. The macro programming support by this framework will lessen the burden on IoT programmers.

6.4.6 Dripcast

Dripcast [53] is a Java based application development framework to integrate smart devices into cloud computing infrastructure. Further, it is a server less framework for storing and processing Java objects in a cloud environment. These Java objects will be made available on smart things and users can manipulate those objects as if they are local objects. It implements transparent Java remote procedure call and a mechanism to read, store, and process Java objects in a distributed, scalable data store. Under Dripcast, all Java objects have worldwide unique ID. The Dripcast framework consists of four components: Client, Relay, Engine and Store.

1. *Client* is a Java library, which works on devices such as smart phones and tablet. It monitors the Java object on the client devices, and forwards remote procedure calls, which are abstracted from the users to the Relay.

2. *Relay* is a stateless, distribution gateway. It forwards requests from clients to corresponding engine servers. A relay server knows the association of object's unique ID and engine servers; a Distributed Hash Table (DHT) manages this association.

3. *Engine* is a set of engine servers. Each engine server runs JavaVM and executes Java methods of an object for a remote procedure call request forwarded by the relay and returns the result back to the relay. If there is a state change of the object, then the new state is stored back into the Store.

4. *Store* is a scalable distributed data-store for storing Java objects with capability for replication and automatic recovery.

The Dripcast framework enables Java based IoT application development.

6.4.7 Calvin

It is a framework [54] that merges IoT and cloud in a unified programming model. It is an IoT programming framework, which combines the ideas of actor model and flow based computing. To simplify application development, it proposes four phases to be followed in a sequential fashion: Describe, Connect, Deploy, and Manage.

- Describe: In this phase, the functional parts of the applications, which are reusable components, are described. In Calvin, everything is treated as an actor: devices, services, and even a piece of computation on cloud. These actors can communicate with each other through ports. To create an actor, a developer describes the actions, their input/output relations, the conditions for a particular action to be triggered, and also the priority between actions. Device manufacturers can supply actors that correspond to their devices as

part of the support code shipped with their devices, enabling their devices to easily integrate with a Calvin application.

- Connect: Once the actors have been described, the next step is to connect those actors by directed graphs between the ports of a number of actors.

- Deploy: In this phase, an application is instantiated according to the graphs provided with its description. The description/connect phase does not specify where the various actors should execute, nor how the data should be transported between them. This is handled during deployment of the

- application. The distributed runtime present at the nodes where the application gets deployed shoulder this responsibility. By forming a mesh network of run time on nodes, actors in a running application can migrate from one runtime to another. Once the runtime has been instantiated and connected the actors locally, the distributed execution environment can move actors to any accessible runtime based on resources, locality, connectivity, and performance requirements.

- Manage: In this phase, the distributed execution environment monitors the applications, handling migration of actors, updates, error recovery and scaling along with book-keeping.

These phases are supported by the run time, APIs and communication protocols. The platform dependent part of Calvin runtime manages communication between runtimes, transport layer support, the inter-runtime communication, and abstraction for I/O, sensing mechanisms to the upper levels of the runtime. The platform independent runtime provides interface for the actors. The scheduler of the Calvin runtime resides in this layer. Calvin runtime supports multi-tenancy. Once an application is deployed, actors may share runtime with actors from other applications.

6.4.8 Simurgh

Simurgh [55] provides a high level-programming framework for IoT application development. The framework supports exposing of IoT services as RESTful APIs and also to compose those IoT services to create various flow patterns in a simplified manner. The overall Simurgh architecture has two main layers: Things layer and Platform layer. In the Thing layer there is a software component called Network Discovery and

Registration Broker, which listens to the incoming connection requests from devices and handles them. There is a rich set of libraries providing device specific interfaces. An API mediator assists programmers to expose their applications through RESTful APIs. Also, they provide RESTful wrappers for those low level device interfaces which are not supported by native vendors, and finally an API manager which monitors APIs access from the external world.

The platform layer has the following components:

1. Thing Description Repository: This stores information about things and services offered by them; periodically updated by the Network Discovery and Registration Broker and API Mediator component. Things are described using TDD (Thing Description Document). A TDD file consists of mainly two parts:

    - Entity Properties: Usually a user-chosen name, last modification date and entity's location is stored.
    - Entity Services: For each of the entity described earlier, entity services define APIs that are available on the entity. These API definition files can be in RESTful API Modeling Language (RAML)[3] or in Swagger[4] format.

2. Two-Phase Discovery Engine: This is used to discover an entity and its corresponding APIs in two phase. In the first phase, the engine will search in TDD repository to find entities based on given criteria. If the goal is finding an API

[3]http://raml.org/
[4]http://swagger.io/

of an entity capable of doing a certain task, then, another search is performed on their respective API Description Documents.

3. Flow Design: This component assists in designing flows, which are chains of IoT services. Through this component, users can discover things, discover their APIs,

   and also can call the found APIs, thereby generating a flow.

4. Flow Composition: Two or more flows can be combined to build a new flow that can deliver a new functionality. This component performs those compositions.

5. Flow Execution Engine: This engine provides all the required resources during the execution of a flow. It configures them and executes all the necessary APIs to fulfill a request.

6. Flow Template Management and Repository: Flows are managed and also to promote flow reusability, the used patterns are stored and are exposed to users when they are designing new flows.

7. Request Management: This component performs user request matching to flow templates. If a match is not found, then the request will be forwarded to Flow Composition module to match with composed flow patterns. Furthermore, if a flow is not found, then users can build the required flow using a flow design interface.

The Simurgh framework provides detailed support to IoT development. Assistance to develop, manage and reuse flow patterns as provided in this framework is crucial for IoT programmers.

6.4.9 High-level Application Development for the Internet of Things

The authors [56] propose a detailed framework for developing IoT applications. They propose a new developmental methodology and a framework to support it. To sim-

plify the process of IoT application development, this framework stresses on identifying stakeholders and demarcating their responsibilities. They can be domain experts, software designers, application developers, device developers and network managers.

In this framework a conceptual model, which serves as a knowledge base about a problem is built taking into account four different areas of concern in IoT application development: domain specific concepts, functional specific concepts, deployment specific concepts, and platform specific concepts.

- *Domain Specific Concepts*: The concepts in this category are unique to an application domain. For example, building automation is reasoned in terms of rooms and floors. There can be sensors, actuators, and storage devices too. These concepts are identified under: Entity of Interest (EoI) which can be any object (e.g., room, book, plant), Resources like sensors, actuators, and storage devices; and the Region used to specify the location of a device.

- *Functionality Specific Concepts*: These concepts describe computational elements of an application and interactions among them. These computational elements are software components that encapsulate and hide a subset of system's functionality and data. Interactions among software components happen through request/response, publish/subscribe and command mode.

- *Deployment Specific Concepts*: These concepts describe information about devices. A device is an entity that provides resources the ability to interact with each other. Each device can host zero or more resources and is located in a region.

- *Platform Specific Concepts*: These are computer programs that act as a translator between hardware devices and an application. They are

categorized as Sensor driver, Actuator driver, Storage Services, and End-user application.

The developmental framework consists of modeling languages and automation techniques to support stakeholders to implement the conceptual model.

- *Support for Domain Concerns*: The developmental framework supports domain concerns in specifying the domain vocabulary using Srijan Vocabulary Language and compiling those vocabulary specifications. This compiled output supports the later phases.
- *Functional Concerns*: For this phase the developmental framework supports by specifying application architecture using Srijan Architecture Language, then compiling the architecture specification, and finally by implementing the application logic
- *Deployment Concerns*: The framework specifies the target deployment of devices using the Srijan Deployment Language, and maps a set of computational services to a set of devices.
- *Platform Concerns*: Here, the device drivers are implemented, the linker generates packages that can be deployed on devices. It basically combines outputs of all the preceding phases like application logic, and device drivers. Device specific code is generated in this phase.

This framework takes the software engineering approach to IoT development and supports model driven development.

6.4.10 PatRICIA

PatRICIA [9] is a programming framework for IoT application development on Cloud platforms. The key feature of this framework is the "intent" based programming model. The programmers can specify the intent and the scope of the intent. Intents can be either a monitoring task on devices or a controlling task of devices. The intent scope delimits the range of an intent. It is the responsibility of the framework to execute the intent on the devices demarcated by the scope of the intent. This programming model, hides many of the underlying complexities of IoT programming from the end users. For example, if PatRICIA is being used in traffic management, then the end user, can simply say "track all the vehicles exceeding the speed limit

90". Here, PatRICIA executes the intent: track the vehicles, on the scope of the intent: all those vehicles, which exceed the speed limit 90 kmph.

The architecture of the framework is 4 tiered. The topmost layer is named as Development Support Layer. It contains tools to aid in application development life cycle. It has a module called Application Manager whose responsibility is to configure, deploy, and license applications along with providing a testing environment for IoT applications. The important part of this layer is that it exposes the programming model based upon "intent" to developers. The Cloud System Runtime layer, provides support for intent based programming by executing the intent on the "scope of the intent". Data and Device Integration layer is responsible for data Management, IoT devices management and virtualization, The Device Communication layer implements different connectors catering to heterogeneous devices. The physical layer has all the things, which can communicate through the Internet.

Intent based programming model: This programming model provides tools to work with monitoring and control tasks. Control tasks help developers to operate, provision, and manage low-level components. They provide a high-level representation of underlying devices and their functionality. They are named "ControlIntent". Likewise monitor tasks, named as "MonitorIntent" are used to subscribe for events from the physical environment along with obtaining and provisioning devices' context. These tasks can be represented as "intents" by application developers, which gets automatically instantiated for the supplied intent scope.

Intent is a data structure representing a specific task, which can be performed in a physical environment. Based on the specified intent, a suitable task is selected (control or monitor), instantiated and executed on the Cloud platform. The Intent, thereby gets translated as a sequence of steps to process data or to perform some actuation on the underlying things. To subscribe to an event in the underlying physical environment or to perform some IoT control, developers can define and configure intents. This shields the developers from the inherent complexity of the IoT.

Intent scope is an abstraction of a group of physical entities which have some common properties. The demarcation of the physical layer for an intent scope is

determined on the Cloud. By specifying the properties that has to be satisfied by physical entities to be in a scope, developers define Intent Scope. PatRICIA also provides operators like send, notify, poll, delimit to work with intents.

The support for intent based programming in PatRICIA will hide many of the underlying heterogeneity, which is advantageous in IoT programming.

6.5 Summary

Each of the programming frameworks discussed earlier have their own advantages in application development. We summarize their key features in Table 6.1.

JAMScript in comparison with other programming frameworks

ELIoT basically extends the Erlang language to cloud-scale IoT programming. Support for broadcast communication, REST interfaces and simulator support are some of the important extensions in ELIoT. JAMScript proposes to use lightweight communication protocols like CoAP for communication, thereby reducing the burden on resource constrained devices. Along with uni-cast messaging, JAMScript also has mechanisms to support broadcast messaging through various composition operators. Calvin combines the ideas of actor model and flow based computing to merge Cloud and IoT programming. The runtime supports multi-tenancy; migration of actors to different runtimes based on spatial locality and constraints. In JAMScript, a thing can be supported for multi-tenancy even when cloud connectivity is not available. Those things which are resource capable can have a local J node. This node can coordinate secure transactions on the thing without any cloud support and once the cloud is available the thing can get hooked onto a J node in the cloud. Mobile Fog extends Cisco's Fog computing model by supporting application development on distributed heterogeneous devices and dynamic scaling through standard APIs. Fog Computing model is advantageous for latency sensitive applications. JAMScript supports latency sensitive applications by allowing J nodes to reside near the edges. Also, the run time of JAMScript ensures load balancing by spawning new J nodes

or merging superfluous J nodes. In the Compose API platform, things are exposed as service objects accessible through RESTful APIs into the clouds. JAMScript also exposes the C nodes to the J nodes. However, the interface is specific to each program depending on what the programmer has implemented in the activities. PYOT is a programming framework for integrating wireless sensor nodes (WSNs) with the cloud. Applications can use sensing and actuating capabilities of motes, shared with the external world through URIs. It enables "in-network processing". JAMScript not only supports edge processing but also provides wide-ranging coordination patterns. In Patricia, a notion called "intent" based programming is supported to focus on computations over a certain elements of the CoT. It provides mechanism for big data management and analytics in the clouds. JAMScript also provides similar capabilities but with stronger coordination primitives. The "predicates" in JAMScript can demarcate the scope of an activity invocation similar to their "intent scope." Dripcast is a server-less Java based application development framework to integrate smart devices into clouds. The framework is used for processing and storing Java objects in a cloud environment. Remote procedure calls on Java objects are implemented in a transparent manner. This framework is quite advantageous to Java based CoT application development. As of today huge number of embedded "things" run C/C++ (for speed and lean footprint) based applications; JAMScript exposes these applications to the cloud resident JavaScript programs, thereby combining the two dominant languages. The data flow framework is as important as the others which are generally based on control flow. IoT data is huge and varied. Data flow approach is seen as one of the possible approaches to handle the complexity of random and huge data. In JAMScript, "Live Variables" tend to handle the control based on data flow.

Framework	**Approach**	**Key features**	**Program's target**	**Coordination support**
Mobile Fog	Macro-programming	Edge processing, dynamic scaling, cloud support, Runtime API support.	Devices	Coordination support through special APIs.
ELIoT	Macro-programming	Extends Erlang for IoT, Support for broadcast communication, RESTful API support, simulator, EVM support.	Devices	Coordination support through Erlang language.
Compose API	Macro-programming	RESTful APIs to access things, cloud support, Composing of services through APIs.	Cloud and devices	Coordination support on Cloud.
Distributed Dataflow support for IoT	Node centric with data flow support	Data flow based IoT application development, edge processing.	Devices	Coordination through choreography.
PyoT	Macro-programming	Edge processing, Python support, URIs for nodes, RESTful APIs.	Devices and web	Coordination support through macro-programming.
Dripcast	Model driven development (Java)	Services in terms of Java Objects, remote management of objects, Cloud support.	Devices	No explicit coordination support.
Calvin	Model driven development	Actor model and data flow based development, cloud support, runtime multi-tenancy support for things.	Devices	Coordination through choreography.
Simurgh	Macro-programming	RESTful API support, flow design and composition support with reusability.	Devices	Orchestration support for flow patterns.
High-level application development for IoT	Model driven development (own languages)	Complete application development life cycle support, division of responsibilities between stakeholders, new languages for vocabulary, architecture and deployment specification.	Devices	Coordination support specified during identification of functional concerns.
Patricia	Macro-programming	Intent based programming, cloud support for control and monitoring of tasks.	Devices and Cloud	Coordination support on cloud, specified through scope of the intent.

Table 6.1 – Summary of programming frameworks

Chapter 7

Conclusions and Future Work

It is estimated by Cisco that only 2% of the present devices are Internet enabled. As more devices get connected, there arises a need for integrating the services provided by these devices into a bigger realm. Programming frameworks should support development of new applications by integrating these services in a simple manner. We believe this can revolutionize the field of future Internet application development. Also, these applications will be geographically distributed over a wide area spanning some hostile conditions, warranting fault tolerance and edge processing for latency sensitive applications. New development frameworks should lessen the impact of application development challenges on programmers. Also, due to huge number of devices, the volume of data available will be astronomical, which can bring in Cloud computing as a required back-end for data management and analytics. Programming frameworks that can unify IoT and Cloud as a single programming model will be advantageous from the developers' point of view.

JAMScript tries to satisfy most of the requirements that we have identified for CoT programming frameworks. It provides *coordination* between Javascript and C programs running on *heterogeneous* platforms. It supports *fault tolerance* through a combination of J and S nodes. The language runtime of C is quite lightweight and also, the advent of lightweight JavaScript engines, like JerryScript[1] guarantees

[1]http://samsung.github.io/jerryscript/

a *lighter footprint* of JAMScript for applications running on constrained devices. J nodes running near the edges takes care of *latency sensitive applications*. We believe JAMScript opens up CoT in a simplified way to the already trained programmers of C and JavaScript for development of new cloud scale applications.

As part of the future work, the JavaScript side of JAMScript including composition operators, should be implemented. The design decisions for healing J nodes to Cloud (once Internet connectivity is available), scaling J nodes based on the available load, and secure exchange of procedures between C and J nodes has to be finalised. Applications need to be built on CoAP and lightweight RPCs, and evaluated for performance. According to the World Economic Forum – 2015, report, Industrial Internet of Things (IIoT), will bring in new opportunities in operational efficiency, outcome economy fuelled by software-driven services, connected ecosystems, and collaboration between humans and machines. Softwares developed using JamScript can be evaluated for adaptability in IIoT domain.

In this thesis, we reviewed the challenges and some of the essential features for IoT/CoT application development which acts as guidelines for the design of our JAMScript framework. Message passing models and embedded language features that are necessary for CoT were explored. The relevance of coordination languages and polyglot programming in CoT was discussed. The section on programming frameworks highlights the programming approaches and key features in some of the frameworks that have been developed for IoT-Cloud application development. The JAMSCript programming framework we presented tries to answer many of the challenges faced by CoT application developers and also simplifies the programming effort required. Since the CoT domain itself is new, the standardization of frameworks will require continuous effort from the CoT community.

Appendix A

BNF for JAMScript

⟨*program*⟩ ::= ⟨*c_program_block*⟩
| ⟨*js_program_block*⟩

A.1 C side

⟨*c_program_block*⟩ ::= ⟨*function_defs*⟩

⟨*function_defs*⟩ ::= ⟨*activity_def*⟩
| ⟨*activity_def*⟩ ⟨*function_defs*⟩

⟨*activity_def*⟩ ::= ⟨*sync_activity*⟩
| ⟨*async_activity*⟩

⟨*sync_activity*⟩ ::= ⟨*c_sync_activity*⟩
| ⟨*js_sync_activity*⟩

⟨*async_activity*⟩ ::= ⟨*c_async_activity*⟩
| ⟨*js_async_activity*⟩

⟨*c_sync_activity*⟩ ::= ⟨*jamd_sync_decl*⟩ ⟨*c_compound_block*⟩

⟨*js_sync_activity*⟩ ::= ⟨*jamd_sync_decl*⟩ ⟨*js_compound_block*⟩

⟨*c_async_activity*⟩ ::= ⟨*jamd_async_decl*⟩⟨*c_compound_block*⟩
⟨*oncomplete_js_compound_block*⟩⟨*onerror_js_compound_block*⟩
| ⟨*jamd_async_decl*⟩⟨*c_compound_block*⟩
⟨*oncomplete_js_compound_block*⟩⟨*onerror_js_compound_block*⟩
⟨*oncancel_c_compound_block*⟩
| ⟨*jamd_async_decl*⟩⟨*c_compound_block*⟩
⟨*oncomplete_js_compound_block*⟩⟨*onerror_js_compound_block*⟩
⟨*onverify_c_compound_block*⟩
| ⟨*jamd_async_decl*⟩⟨*c_compound_block*⟩
⟨*oncomplete_js_compound_block*⟩⟨*onerror_js_compound_block*⟩
⟨*oncancel_c_compound_block*⟩⟨*onverify_c_compound_block*⟩
| ⟨*jamd_async_decl*⟩⟨*c_compound_block*⟩
⟨*oncomplete_js_compound_block*⟩⟨*onerror_js_compound_block*⟩
⟨*onverify_c_compound_block*⟩⟨*oncancel_c_compound_block*⟩

⟨*js_async_activity*⟩ ::= ⟨*jamd_async_decl*⟩⟨*js_compound_block*⟩
⟨*oncomplete_c_compound_block*⟩⟨*onerror_c_compound_block*⟩

⟨*jamd_sync_decl*⟩ ::= `"jamdef"` `"sync"` ⟨ *c_func_decl*⟩
| `"jamdef"` `"sync"` ⟨ *c_func_decl*⟩ `"in"` ⟨*identifier*⟩
| `"jamdef"` `"sync"` ⟨ *c_func_decl*⟩ `"requires"` ⟨*identifier*⟩
| `"jamdef"` `"sync"` ⟨ *c_func_decl*⟩ `"in"` ⟨*identifier*⟩ `"requires"`
⟨*identifier*⟩

⟨*jamd_async_decl*⟩ ::= `"jamdef"` ⟨ *c_func_decl*⟩
| `"jamdef"` ⟨ *c_func_decl*⟩ `"in"` ⟨*identifier*⟩
| `"jamdef"` ⟨ *c_func_decl*⟩ `"requires"` ⟨*identifier*⟩
| `"jamdef"` ⟨ *c_func_decl*⟩ `"in"` ⟨*identifier*⟩ `"requires"` ⟨*identifier*⟩

⟨*c_compound_block*⟩ ::= `"{"` ⟨*c_statements*⟩ `"}"`

⟨*js_compound_block*⟩ ::= `"{"` ⟨*js_statements*⟩ `"}"`

⟨*oncomplete_js_compound_block*⟩ ::= ⟨*oncomplete_decl*⟩⟨*js_compound_block*⟩

⟨*oncomplete_c_compound_block*⟩ ::= ⟨*oncomplete_decl*⟩⟨*c_compound_block*⟩

⟨*onerror_js_compound_block*⟩ ::= ⟨*onerror_decl*⟩⟨*js_compound_block*⟩

⟨*onerror_c_compound_block*⟩ ::= ⟨*onerror_decl*⟩⟨*c_compound_block*⟩

⟨*oncancel_c_compound_block*⟩ ::= ⟨*oncancel_decl*⟩⟨*c_compound_block*⟩

⟨*onverify_c_compound_block*⟩ ::= ⟨*onverify_decl*⟩⟨*c_compound_block*⟩

⟨*oncomplete_decl*⟩ ::= `"oncomplete"` ⟨*c_func_decl*⟩

⟨*onerror_decl*⟩ ::= `"onerror"` ⟨*c_func_decl*⟩

⟨*oncancel_decl*⟩ ::= `"oncancel"` ⟨*c_func_decl*⟩

⟨*onverify_decl*⟩ ::= `"onverify"` ⟨*c_func_decl*⟩

⟨*identifier*⟩ ::= /* inherited from c grammar. */

⟨*c_statements*⟩ ::= /* inherited from c grammar. */

⟨*c_func_decl*⟩ ::= /* inherited from c grammar */

⟨*js_statements*⟩ ::= /* inherited from ES5 grammar */

Bibliography

[1] Diane J Cook and Sajal K Das. How smart are our environments? An updated look at the state of the art. *Pervasive and Mobile Computing*, 3(2):53–73, 2007.

[2] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, September 2013.

[3] Dominique Guinard, Vlad Trifa, Friedemann Mattern, and Erik Wilde. From the Internet of Things to the Web of Things: Resource-oriented Architecture and Best Practices. *Architecting the Internet of Things*, pages 97–129, 2011.

[4] Mohammad Aazam, Imran Khan, Aymen Abdullah Alsaffar, and Eui-Nam Huh. Cloud of things: Integrating internet of things and cloud computing and the issues involved. In *Proceedings of the 11th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, pages 414–419. IEEE, 2014.

[5] Colin Dixon, Ratul Mahajan, Sharad Agarwal, A J Bernheim Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. An Operating System for the Home. *NSDI*, pages 337–352, 2012.

[6] Stephen Dawson-Haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev, and David E Culler. BOSS: Building Operating System Services. *NSDI*, pages 443–457, 2013.

[7] Debashish Ghosh, Muthucumaru Maheswaran, and Fan Jin. JADE: A Unified Programming Framework for Things, Web, and Cloud. In *Proceedings of the International Conference on Internet of Things*, pages 1–6, April 2014.

[8] Rajkumar Buyya, Toni Cortes, and Hai Jin. Single System Image. *IJHPCA*, 15(2):124–135, 2001.

[9] Stefan Nastic, Sanjin Sehic, Marko Vogler, Hong-Linh Truong, and Schahram Dustdar. Patricia–a novel programming model for iot applications on cloud platforms. In *Proceedings of the IEEE 6th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 53–60. IEEE, 2013.

[10] Alessio Botta, Walter de Donato, Valerio Persico, and Antonio Pescapé. On the integration of cloud computing and internet of things. In *Proceedings of the International Conference on Future Internet of Things and Cloud (FiCloud),2014*, pages 23–30. IEEE, 2014.

[11] George A Papadopoulos and Farhad Arbab. Coordination Models and Languages. pages 329–400. Elsevier, 1998.

[12] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96, 1992.

[13] The top programming languages-ieee spectrum. `http://spectrum.ieee.org/static/interactive-the-top-programming-languages`. Accessed: 3-September 2015.

[14] Michael Barr. *Programming embedded systems in C and C++*. O'Reilly Media, Inc., 1999.

[15] David Gay, Philip Levis, Robert Von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Acm Sigplan Notices*, volume 38, pages 1–11. ACM, 2003.

[16] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. *Ambient intelligence*, 35, 2004.

[17] Cx51 user's guide: Language extensions. `http://www.keil.com/support/man/docs/c51/c51_extensions.htm`. Accessed: 3-September 2015.

[18] Ahmed Amine Jerraya, Sungjoo Yoo, Diederik Verkest, and Norbert Wehn. *Embedded software for SoC*. Springer, 2003.

[19] Brinch Hansen. The design of edison. *Software: Practice and Experience*, 11(4):363–396, 1981.

[20] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[21] Andreas Reinhardt, Parag S Mogre, and Ralf Steinmetz. Lightweight remote procedure calls for wireless sensor and actuator networks. In *Proceedings of the International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, pages 172–177. IEEE, 2011.

[22] Brian N Bershad, Thomas E Anderson, Edward D Lazowska, and Henry M Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems (TOCS)*, 8(1):37–55, 1990.

[23] Kamin Whitehouse, Gilman Tolle, Jay Taneja, Cory Sharp, Sukun Kim, Jaein Jeong, Jonathan Hui, Prabal Dutta, and David Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *Proceedings of the 5th international conference on Information processing in sensor networks*, pages 416–423. ACM, 2006.

[24] Terry D May, Shaun H Dunning, George A Dowding, and Jason O Hallstrom. An rpc design for wireless sensor networks. *International Journal of Pervasive Computing and Communications*, 2(4):384–397, 2007.

[25] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[26] Xinyang Feng, Jianjing Shen, and Ying Fan. Rest: An alternative to rpc for web services architecture. In *Proceedings of the First International Conference on Future Information Networks, 2009. ICFIN 2009.*, pages 7–10. IEEE, 2009.

[27] Cesare Pautasso. Composing restful services with jopera. In *Software Composition*, pages 142–159. Springer, 2009.

[28] Justin R Erenkrantz, Michael Gorlick, Girish Suryanarayana, and Richard N Taylor. From representations to computations: the evolution of web architectures. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 255–264. ACM, 2007.

[29] Matthias Kovatsch, Simon Duquennoy, and Adam Dunkels. A low-power coap for contiki. In *Proceedings of the 8th International Conference on Mobile Adhoc and Sensor Systems (MASS)*, pages 855–860. IEEE, 2011.

[30] Alessandro Ludovici, Pol Moreno, and Anna Calveras. Tinycoap: a novel constrained application protocol (coap) implementation for embedding restful web services in wireless sensor networks based on tinyos. *Journal of Sensor and Actuator Networks*, 2(2):288–315, 2013.

[31] Tapio Levä, Oleksiy Mazhelis, and Henna Suomi. Comparing the cost-efficiency of coap and http in web of things applications. *Decision Support Systems*, 63:23–38, 2014.

[32] George Wells. Coordination languages: Back to the future with linda. In *Proceedings of WCAT'05*, pages 87–98, 2005.

[33] David Kitchin, Adrian Quark, William R. Cook, and Jayadev Misra. *The Orc Programming Language*, volume 5522 of *Lecture Notes in Computer Science*. Springer, 2009.

[34] William R Cook, Sourabh Patwardhan, and Jayadev Misra. Workflow patterns in orc. In *Coordination Models and Languages*, pages 82–96. Springer, 2006.

[35] Fabrizio Montesi, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Jolie: a java orchestration language interpreter engine. *Electronic Notes in Theoretical Computer Science*, 181:19–33, 2007.

[36] Fabrizio Montesi, Claudio Guidi, Ivan Lanese, and Gianluigi Zavattaro. Dynamic fault handling mechanisms for service-oriented applications. In *Proceedings of the Sixth European Conference on Web Services*, pages 225–234. IEEE, 2008.

[37] H Fjeldberg. Polyglot Programming: A business perspective. Master's thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2008.

[38] Juhana Harmanen. Polyglot Programming in Web Development. Master's thesis, Tampere University Of Technology, Finland, 2013.

[39] Ola Bini. Fractal programming. `https://olabini.com/blog/2008/06/fractal-programming/`. Accessed: 3-September 2015.

[40] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[41] Michael Treaster. A survey of fault-tolerance and fault-recovery techniques in parallel systems. *arXiv preprint cs/0501002*, 2005.

[42] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *Computer*, (4):68–74, 1997.

[43] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, volume 39, pages 111–122. ACM, 2004.

[44] Douglas C Schmidt, Aniruddha Gokhale, and Balachandran Natarajan. Frameworks: Why they are important and how to apply them effectively. *ACM Queue magazine*, 2(5), 2004.

[45] Sunil Jardosh and Pankesh Patel. Application development approaches for the internet of things: A survey. In *Proceedings of the IEEE Conference- TENSYMP*, 2015.

[46] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[47] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. Mobile fog: A programming model for large–scale applications on the internet of things. *Network*, 12(F13):F14, 2013.

[48] Alessandro Sivieri, Luca Mottola, and Gianpaolo Cugola. Drop the phone and talk to the physical world: Programming the internet of things with erlang. In *Proceedings of the Third International Workshop on Software Engineering for Sensor Network Applications*, pages 8–14. IEEE Press, 2012.

[49] Alessandro Sivieri. *ELIoT: A Programming Framework for the Internet of Things*. PhD thesis, Politecnico di Milano, Italy, 2014.

[50] Juan Luis Pérez, Álvaro Villalba, David Carrera, Iker Larizgoitia, and Vlad Trifa. The compose api for the internet of things. In *Proceedings of the companion publication of the 23rd international conference on World wide web companion*, pages 971–976. International World Wide Web Conferences Steering Committee, 2014.

[51] Michael Blackstock and Rodger Lea. Towards a distributed data flow platform for the web of things. In *Proceedings of the 5th International Workshop on Web of Things*. IEEE, 2014.

[52] Andrea Azzara, Daniele Alessandrelli, Matteo Petracca, and Paolo Pagano. Demonstration abstract: Pyot, a macroprogramming framework for the iot. In *Proceedings of the 13th international symposium on Information processing in sensor networks*, pages 315–316. IEEE Press, 2014.

[53] I. Nakagawa, M. Hiji, and H. Esaki. Dripcast – server-less java programming framework for billions of iot devices. In *Proceedings of the 38th International Computer Software and Applications Conference Workshops (COMPSACW)*, pages 186–191, July 2014.

[54] Per Persson and Ola Angelsmark. Calvin-merging cloud and iot. In *Procedia Computer Science*. Elsevier, 2015.

[55] Farzad Khodadadi, Amir Vahid Dastjerdi, and Rajkumar Buyya. Simurgh: A framework for effective discovery, programming, and integration of services exposed in iot. In *Proceedings of the International Conference on Recent Advances in Internet of Things (RIoT)*, pages 1–6. IEEE, 2015.

[56] Pankesh Patel and Damien Cassou. Enabling high-level application development for the internet of things. *CoRR*, abs/1501.05080, 2015.