Design and Implementation of a Compiler for a Fog and IoT Programming Language

Robert Wenger

School of Computer Science
McGill University
Montreal, Canada

December 2018

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

2018/12/14

Abstract

The growing ubiquity of internet-connected devices has led to increased interest in the Internet of Things. As the pace of IoT development accelerates, new paradigms are being developed that can fully utilize the potential benefits of this new computing trend. One such paradigm is fog computing, in which data processing is moved toward the edge, allowing more localized devices to handle tasks that would have previously been assigned to the cloud. In this thesis we introduce JAMScript, a new programming language specifically designed for fog computing. JAMScript combines the C and JavaScript programming languages, allowing the ability to leverage the benefits of each language in a single program. JAMScript takes advantage of this shared language space to provide edge computing optimized functionality, such as remote multi-language activities, shared persistent variables, conditional execution constructs and data stream management. We explain the design and implementation of the JAMScript compiler and provide details about how applications can be developed using JAMScript. We then provide example programming patterns where JAMScript's unique features allow building complex applications with minimal effort.

Résumé

L'omniprésence croissante des appareils connectés à Internet a suscité un intérêt croissant pour l'Internet des objets. Comme le développement d'IdO s'accélère, de nouveaux paradigmes en cours de développement peuvent tirer parti des avantages potentiels de cette nouvelle tendance informatique. L'un de ces paradigmes est l'informatique de brouillard; Le traitement des données est déplacé vers le bord, ce qui permet aux dispositifs plus localisés de gérer les tâches qui auraient été normalement assignées au nuage. Au cours de cette thèse, nous introduisons un nouveau langage de programmation spécialement conçu pour l'informatique de brouillard, JAMScript. JAMScript combine les langages de programmation C et JavaScript, permettant la possibilité de tirer parti des avantages de chaque langue dans un seul programme. JAMScript profite de cet espace total de langage pour fournir des fonctionnalités optimisées de traitement informatique, telles que les activités multilingues à distance, les variables permanentes partagées, les utilitaires de construction d'exécution et la gestion des flux de données. Nous expliquons la conception et la mise en œuvre du compilateur JAMScript et fournissons des détails sur la manière dont les applications peuvent être développées à l'aide de JAMScript. Nous poursuivrons en présentant quelques exemples de schémas de programmation où les fonctionnalités uniques de JAMScript permettent de créer des applications complexes avec un effort minimal.

Acknowledgments

I would like to thank Professor Muthucumaru Maheswaran for his mentorship and guidance during past few years, without who I would not have been able to complete the thesis writing process.

I would also like to thank David Echomgbe, Jayanth Krishnamurthy, Lilly Jiang, Jian-hua Li, Owen Li, Carl Liu, Keith Strickling, Rossen Vladimirov, Nicolas Truong and Xiru Zhu for their contributions to JAMScript.

Lastly, I would like to thank my family for all their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Internet of Things (IoT) is ushering in a new paradigm of computing. In this paradigm, IoT devices generate large volumes of data at very high rates, requiring computations to be performed in a timely manner to extract intelligence from the data. The extracted intelligence, along with other contextual information, could be used to control the overall operation of the IoT. Due to the large volume of data, IoT requires high capacity computing backends to deal with data processing. One way of satisfying this requirement is to use cloud computing, which can elastically adjust the computing capacities to precisely match the computing demands created by the IoT. Although cloud computing is an ideal backend for many IoT scenarios, it places the computing resources at distant data centers, which creates long latencies between the IoT devices that generate the data and the eventual processing nodes. In particular, cloud computing as a computing backend is not suitable for IoT applications that need to process captured data within a short time delay. To address the latency issue fog computing has been recently developed. In fog computing the computing resources are placed at the edge of the Internet such that they are closer to the IoT devices.

IoT is quite diverse and can be categorized into two major configurations: with and without mobility. For instance, smart building is an IoT scenario without mobility where the devices can be connected to a cloud-based computing backend with fog computing providing low-latency resource pools for computation offloads. Smart cars are IoT where mobility is a major concern. Smart cars could use fogs for low-latency compute offload, as well as contextual data sharing so that repeat processing of the same data could be minimized.

With distributed computing reliability is an important concern. Loss of computational state due to failures will hinder the forward progress of computations. The traditional method of implementing reliability in distributed computing systems is performed by replicating the computational state across machines that do not share common modes of failures. Cloud computing has become a dominant host for many popular applications because replication-based reliability schemes have succeeded in masking the failures to an acceptable level. In fog computing, it is not possible to efficiently implement the same core reliability enhancing mechanisms that are used with cloud computing systems because fogs are distributed with each site holding a small number of machines. Further, the machines in a single site could have highly correlated failures because they are deployed close to the edge, far away from highly redundant and sophisticated installations such as data centers. Selecting fog machines from different sites for backing up with each other will defeat the primary purpose of fog computing, to provide low latency processing.

1.1 The JAMScript Language

JAMScript is a language and runtime developed to tackle the challenge of developing resilient applications for a collective of devices, fogs, and cloud. JAMScript is heavily inspired by the recent trend of developing programming language integrated mechanisms for reliable computing over cloud resources or intermittent computing systems. For example, Resilient Distributed Datasets (RDDs) [1] are the building blocks of the highly popular Apache Spark framework. Similarly, Chain [2] is an extension of C and a runtime library for efficiently harvesting idle cycles from intermittently available computing resources.

JAMScript is expressly designed to meet the following objectives while enabling distributed computing over a collective of devices, fogs, and cloud:

- Disconnection tolerance: IoT Devices must retain their functionality even when network disconnections occur. One way of meeting this requirement is to ensure that devices in each network partition have all the necessary components to function. The JAMScript components remaining in a partition must be able to discover each other and reorganize themselves into a functioning whole.

- Efficient synchronization and orchestration: IoT often refers to relatively tiny devices that are more interesting as a collective than taken individually. To control the operation of IoT as a collective, we need mechanisms that can efficiently synchronize the collective. Using synchronization, we can launch operations across the collective at the same time and ensure the devices act on the physical environment at the same time. Another important aspect of orchestration is partitioning the collective into subgroups and making the different subgroups perform different actions.

- Reliable state management: Cyber-physical systems such as IoT have unique challenges with regard to reliable state update. In IoT, the process state can be linked to the state of the physical device in different ways. When a failure occurs, recovering the old state and resuming processing will be heavily dependent on the application scenario. For example, a failed drone in a swarm of drones needs to quickly rejoin the swarm's current position after resuming operation instead of continuing its processing from the point where it failed. Also, in many deployments system-level fault tolerance measures such as triple modular redundancy could be an issue. If a component insists on resuming its processing from where it last failed, it will never agree with the rest of the group because the system state has moved on, ignoring the failed component.

- Node and application scaling: Dealing with a large number of nodes is an important requirement for a language and runtime for IoT. The runtime needs to support scalable collective operations so that the running time does not grow with increasing number of nodes. In addition to the number of nodes, IoTs can host complex applications. Complex applications can be decomposed into several interoperating programs and their coordinated execution should yield a solution for the application scenario.

A JAMScript program has two types of functions: J (written in JavaScript) and C (written in C). A JAMScript program will be able to run if J type functions are able to locate and call C type functions and vice-versa. For disconnection-tolerant JAMScript program deployment, we need to ensure that any partition resulting from a disconnection has runnable C and J functions. That is, each partition holds J and C type functions so the nodes in the partition are still able to run the JAMScript program. Although a JAMScript program can run in the nodes in the disconnected partition, its functionality could be reduced compared to the functionality the program will have when the system is in a connected state. In JAMScript functions can be tagged to run only in a connected state, while others can be tagged to run only in a disconnected state. Running functions tagged for connected execution from a node in a disconnected partition will result in an execution error. JAMScript programs can use this feature to detect the disconnection state and adapt the processing steps.

An instantiation of a JAMScript program follows a hierarchical structure that corresponds to the cloud, fog, and device hierarchy. A device would run both C and J type functions of the JAMScript program. Because the device is running both types, it is a self-contained execution of the JAMScript program. If the device is connected to the fog, the JAMScript program's components (J and C) running in the device would connect to the corresponding program running in the fog to create a larger instance of the program. Similarly, many devices could run the same JAMScript program and all of them will create a tree formation rooted at the fog. Likewise, if the fog is connected to the cloud and the cloud is running the same JAMScript program an even larger distributed instance of the JAMScript program is formed. It should be noted that the program instances across the machines (cloud, fog, and devices) connect with each other only if they are running instances of the same JAMScript program. This way there are no interoperability problems. A remote function invocation can be interpreted correctly and unambiguously.

The hierarchical structure created by the running instances of a JAMScript program can be a tree rooted at the cloud if all devices are fully connected. If a fog disconnects along with the devices connected to it from the rest of the internet, for example in a moving system like a train, the fog would form a subtree that is disconnected from the global tree of JAMScript instances.

Remote function invocations can be either up tree or down tree. Down tree invocations involve several nodes. For instance, when a fog invokes a remote function all devices connected to the fog receive the request for execution. We have two modes of remote function invocations: synchronous and asynchronous. In synchronous remote function invocations, all connected devices are expected to start their function executions at the same time. The invoking node waits for all nodes (or a quorum) to complete the execution before collecting the results and returning it as the outcome of the invocation. In certain application scenarios, it may be necessary to partition the nodes under the root of the tree based on some attribute and invoke a given function on a specific partition. Many device orchestration patterns can be implemented using a combination of synchronous remote invocations and tree partitioning directives.

Resilient state management to enable fault-tolerant IoT is a major goal of JAMScript design. The devices can fail, leave, move to another location, or arrive anew. The fogs can fail, become unresponsive due to overloading, or become non-performant due to device mobility that increases the distance between the device and fog. The cloud, on the other hand, is assumed to be elastic and reliable. The devices log necessary state information to the fogs. JAMScript provides a persistent storage class for logging information to the fogs. The programmer can use this facility to save important state information to the fogs. The fogs apply functional transformations to the logged information to compute analytics and/or reconstruct a restoration state for the devices. The overriding design concern is not to slow the fog computing operations because fast response times are the primary motivation for fog computing. Therefore, JAMScript relies on some assumptions. One of them is that loss of few updates could be tolerated without catastrophic consequences. The log updates are sent to multiple non-collocated fogs so the impact of common mode failures can be minimized. Also, the transformations performed by the fogs on the logged data are functional. Any loss of data computed by applying the transformations can be recovered by redoing the computations on the logs.

One of the attributes of IoT is the ability to integrate large quantities of devices. Therefore, scalability is an important concern in developing JAMScript. To address this concern JAMScript uses a two-level hierarchy. A number of fogs connect to the cloud at the first level. The fog servers interconnect with the cloud via a publish-subscribe protocol, such as MQTT. This makes the design friendly to fog disconnections and mobility although they may not be frequent. The JAMScript programming model does not directly support fog-to-fog interactions, although the runtime uses such interactions to replicate and restore data for reliability purposes. The second level of the hierarchy interconnects the devices to the fogs. There could be a large number of devices underneath a given fog. The devices also connect to a fog via the MQTT publish-subscribe protocol. Because devices can frequently disconnect from a fog due to either device mobility or fog failure, we have redundant associations between the fog and a device. In fact, the best fog selection for a device is a challenging problem that needs to be solved in the best possible way to obtain the greatest performance from fog computing while retaining the necessary consistency levels. To simplify the programming model, JAMScript exposes a single fog node as a parent per device; this way the logical model is a tree. However, in the physical realization a device can have multiple parent fogs for the necessary fault tolerance.

JAMScript is a single-threaded language. Using asynchronous functions that are supported in the J and C components, many concurrent programming patterns could be implemented. However, for complex concurrent patterns the JAMScript language supports a distributed inter-application data exchange (IAX) facility. Using IAX, an application can share data in its JAMScript managed persistent storage with another application. The data in this storage is created by the devices through data logging or by the computational transformations applied by the fogs on the accumulated data. Although the simplest data exchange through IAX could be between two applications, multiple applications can engage in the data exchange. The data stored in the JAMScript managed persistent storage is immutable. That is, data is appended to a stream when devices write to the store. Similarly, when the functional transformations (called flows) create new data using existing data, new streams are created and data is appended to the store. Because IAX allows efficient data exchange between applications that take place simultaneously at multiple fogs, it is recommended that complex applications are decomposed into smaller tasks with each task implemented via a JAMScript program. The tasks use the IAX facility to collaboratively solve the original application.

1.2 Thesis Contribution

In this thesis we introduce a completed version of JAMScript; a programming designed for fog computing. We present the syntax for the language and describe its features in depth. We design and implement a compiler for JAMScript, explain the compiler's components and provide a guide on how to add new features. We develop multiple programming patterns to show the advantages of using JAMScript in different situations. We then demonstrate the performance of JAMScript through multiple test cases.

1.3 Organization of the Thesis

This document is organized as follows. Chapter 2 provides background research on the Ohm compiler, including a detailed tutorial. Chapter 3 describes the architecture of the components related to the JAMScript compiler. Chapter 4 contains a description and syntax of the features of the JAMScript programming language. Chapter 5 covers the implementation of the JAMScript compiler, including detailed descriptions of how the compiler goes from input code to output. In Chapter 6 we detail possible programming patterns that can be achieved using with JAMScript, including code examples. Chapter 7 presents the results of an examination of the performance of various JAMScript features. Chapter 8 discusses other works in relation to the challenges of JAMScript.

Chapter 2

Background Research

2.1 Using Ohm Parser Generator

The JAMScript compiler is built with the Ohm parser generator [3]. Ohm uses Parsing Expression Grammars (PEGs) [4], a form of top-down parsing where the first rule to detect a match is satisfied immediately. Because of this, PEGs cannot have ambiguity errors such as shift-reduce errors that can happen when writing a Context Free Grammar (CFG), used by many popular compiler generator tools such as ANTLR [5] and Bison [6]. This presents different challenges than those normally found when writing a parser, as the ordering of rules can significantly change the way how an input file is parsed. Different orderings of rules may even be able to validly parse a given input file but with widely varying outputs. This could lead to requiring a step by step examination of how the parser parsed the file to determine why the outputs differ. The design of Ohm allows parser generators to be implemented in many different programming languages, currently there are two implementations: Ohm/JS [7] in JavaScript and Ohm/S [8] in Smalltalk.

An Ohm parser is composed of two parts: grammar files and semantic actions. The grammar files are language implementation independent, while the semantic actions are written in the host language of the implementation. This means that an Ohm grammar written for a JavaScript-based compiler should be equally valid for a Smalltalk based compiler. The JAMScript compiler is written in JavaScript, and this chapter will explain how to use semantic actions in the context of an Ohm/JS implementation.

Ohm Grammars

Ohm grammars are a series of rules that define a language. Each rule is written as an equation where the left-hand side is the name of the rule and the right-hand side is a series of statements that must all be satisfied. Every Ohm grammar begins with a name specified before opening a block containing the rules for grammar.

```
grammarName {
  SyntacticRule = digit+
  lexicalRule = digit+
}
```

Syntactic Rules

Syntactic rules are rules denoted by beginning their name with an uppercase letter. Syntactic rules ignore whitespace characters between the expressions of a rule. Whitespace is represented by a predefined rule called space that can be overridden to redefine what a whitespace character is. In our sample grammar, the rule **SyntacticRule** would match with any string that was made up of one or more digits including those with spaces in the middle:

```
> 0
> 123
> 12 123 12
```

Lexical Rules

Lexical rules begin with a lowercase letter. Lexical rules are whitespace sensitive, if there is a whitespace character (such as a space or new line) in the middle of an input string then the rule will stop matching when it reaches the whitespace. In our sample grammar, the rule **SyntacticRule** would match with any string that was made up consecutive digits with no spaces:

```
2
1324
```

Operators

Ohm supports a number of operators that can be used in conjunction with expressions to alter their functionality.

`expression*`

Matches an expression 0 or more times.

```
rule = letter digit*
> a
> a0
> a123
```

`expression+`

Matches an expression 1 or more times.

```
rule = digit+
> 0
> 1
> 1234
```

`expression?`

Matches an expression 0 or 1 time

```
rule = "-"? digit+
> -1
> 1
> -123
> 123
```

`terminal .. terminal`

Range operator that matches between specified characters.

```
rule = "0" .. "9"
> 0
> 5
> 9
```

`(expression expression)`

Groups together multiple expressions

```
rule = digit+ ("." digit+)?
> 12
> 20.34
```

`expression | expression`

Alteration: Match either of two expressions

```
rule = "true" | "false"
> "true"
> "false"
```

`~expression`

Negative lookahead: Match if not equal to the expression

```
rule = ~"0" digit+
> 100
> 23
```

The rule would match with any number that did not begin with a zero

`&expression`

Lookahead: Match if the expression matches, but do not consume it

```
rule = &"a" letter+
> abc
> aaa
```

The rule `&"a" letter+` will match any character string that begins with an a.

This differs from using `"a" letter*` as using the lookahead operator will return the string as one token, which simplifies using it later, while not using the lookahead operator will generate two tokens.

`#expression`

Lexification: Matches a syntactic rule as if it were a lexical rule.

```
AnyDigit = digit+
```

This rule would match combinations of digits with spaces in the middle. If there are situations we would like to use a rule but want to exclude whitespace we can reference it in other rules by using AnyDigit.

Rules Declaration

When creating rules, it is common to group together multiple rules with similar functionality into one:

```
AlphaNum = Letter
         | Number
```

This creates a rule called AlphaNum with two possible paths for matching.

```
AddExp = AddExp "+" MultExp    -- Add
       | AddExp "-" MultExp    -- Subtract
       | Expr
```

Unnamed branches can only have an arity (number of arguments) greater than one if there are no named branches. In this example, two branches have an arity of three and one branch has an arity of one, so we must use inline rule declarations. This allows us to name each branch and must be done for each branch that has an arity different from the unnamed branches.

Internally this creates three separate rules for `AddExp`:

```
AddExp_Add = AddExp "+" MultExp
AddExp_Subtract = AddExp "-" MultExp
AddExp = AddExp_Add
       | AddExp_Subtract
       | Expr
```

This corrects the arity problem, as all branches of `AddExp` have an arity of one now, while `AddExp_Add` and `AddExp_Subtract` have an arity of two.

Parameterized Rules

Ohm allows writing rules with parameters that take in arguments. For example:

```
Double<x> = x x
```

We can then use the parametrized rule by calling it in a rule:

```
DoublesString = "'" Double<letter>+ "'"
```

Ohm provides built-in parametrized rules that are available in both lexical and syntactical versions. The list of built-in syntactical rules is shown in Table 2.1. Each built-in rule is also available as a lexical rule by changing the first letter to a lower case character.

Table 2.1: Built-in Ohm rules

Rule	Description
`ListOf<element, separator>`	Match with any number of elements separated by a separator or an empty list.
`NonemptyListOf<element, separator>`	Match with any number of elements separated by a separator.
`EmptyListOf<element, separator>`	Empty case generated by ListOf if it does not match with anything.

Rule Descriptions

Ohm allows rules to have an optional description that is used in error messages. These descriptions help make error messages easier to read for the end user when parsing fails.

For example, if we have the rule:

```
alphaNum = "a" .. "z" | "A" .. "Z" | "0" .. "9"
```

If parsing failed at this rule Ohm would output an error message:

```
Expected "0".."9", "A".."Z", or "a".."z"
```

We can add a description to the rule to make the error message more user friendly:

```
alphaNum (an alphanumerical) = "a" .. "z" | "A" .. "Z" | "0" .. "9"
```

If parsing failed at this rule, the error message would be:

```
Expected an alphanumerical
```

Inheritance

We can extend existing grammars by using inheritance. This is done by using the inheritance operator when naming the grammar:

```
newGrammar <: parentGrammar {
  ...
}
```

In this case, **newGrammar** will automatically have access to all the rules of **parentGrammar**. A rule inherited from the parent grammar can be mixed and matched with rules in the current grammar when creating new rules:

```
newRule = inheritedRule+ expression+
```

If the name of a new rule you are trying to create already exists in the parent grammar then this will generate a rule conflict error.

Operators

There are two operators that are exclusive to inherited grammars:

Override

Override a rule in the parent grammar:

```
rule := expression
```

Extend

Extend a rule in the parent grammar as a prepended alteration:

```
rule += expression
```

This will create a new rule that is equivalent to:

```
rule = expression | rule
```

2.2 Example Grammar using Ohm

```
addingGramar {
    Expr = Expr "+" number      -- Add
         | number

    number = digit+
}
```

In the example above we show a simple valid Ohm grammar. The grammar is named **addingGrammar**. The rule `number` is defined as any combination of one or more digits. Because the rule starts with a lower-case letter it is a lexical rule, therefore spaces are not ignored and any spaces between the digits would end the matching before that space. If the rule was written with a capital then spaces could be entered between the digits and it would still match as a single number. The rule `Expr` is defined as either a number or an `Expr` followed by the addition of a `number`. Since this rule is a recursive rule it would match with any length of numbers with plusses in-between.

Examples of valid input:

```
> 424
> 13+823
> 1129 + 123 + 654
```

2.3 Translators in Ohm

Ohm provides a way to generate output code by traversing the tree created by the parser using actions called attributes and operations. The two perform similar functionality and are written the same way. Attributes are memoized, for each individual node in the tree the result is evaluated only once. Operations are not memoized, when you call on a single node multiple times the result will be evaluated each time.

The Ohm parser generates a Concrete Syntax Tree (CST), which is a complete representation of the parsed input. The difference between using a CST compared to an Abstract Syntax Tree (AST) is that the CST contains superfluous information that is not necessary when translating the parse tree.

Operators

Ohm translators are comprised of a collection of semantic actions that map to each expression in the parse tree. Actions are stored in a JavaScript object, where for each expression in the grammar we assign a function to the property with a matching name. The function must have the same number of arguments (arity) that the expression has. When the function is called the arguments to the function will contain the child nodes of the expression.

For example, if we wanted to create a translator for a grammar that contained only the rule:

```
HelloW = "Hello" "World"
```

We would then create a new JavaScript object:

```
var trans = {
  HelloW: function(match1, match2) {
    return(match1.sourceString + " " + match2.sourceString);
  }
}
```

In this example, the argument **match1** would contain information about the characters of `Hello`, and the argument **match2** would contain information about the characters of `World`. Because we don't need to recursively translate on strings where the value is fixed, we can call the `sourceString` property of each child node and get their raw string values. We can also call a syntax action on the arguments of the expression by calling the current translator or another translator. For example, if we declared **trans** as an operation, we could call `match1.translator()` to translate further down the tree. If we declare **trans** as an attribute, we could call `match1.translator` to translate further down the tree. When we have rules that have alternatives, such as:

```
AddExp = AddExp "+" MultExp        -- Add
       | AddExp "-" MultExp        -- Subtract
       | Expr
```

We must define the translation rule for each alternative separately:

```
...
AddExp_Add: function(left, plus, mult) {
    return left.translator + " + " + mult.translator;
},
AddExp_Subtract: function(left, minus, mult){
    return left.translator + " - " + mult.translator;
},
AddExpr: function(expr) {
    return expr.translator;
}
...
```

When traversing through a CST, every item in an expression must be specified as a parameter of the syntax action function. This includes terminal nodes, which in many cases are fixed string values and the input contained in them can be safely ignored. This can be seen in the example above; the plus and minus symbols must have a parameter in the rules where they are being used. As the value inside the parameter is always fixed we can ignore the contents of the variable and use a predefined string instead.

If there is no rule with a matching name, then Ohm will attempt to match with a predefined semantic action. Table 2.2 shows a list and description of the predefined actions. These rules can be overridden to allow a single function to apply to multiple nodes.

Table 2.2: Ohm generic rules

Rule	Description
`_iter`	Used for matching on iteration nodes, such as those produced by using `expression+`, if not set return an array with the result of parsing each iteration node.
`_nonterminal`	Used for matching on nonterminal nodes (nodes with children), if not set and the node only has one child this will return the result of parsing the child node.
`_terminal`	Used for matching on terminal nodes, such as string matches

If a rule does not exist for a node that has only a single child, then the syntax action will automatically be called recursively on that child. If the node has multiple children and no matching rule is found, then the compiler will throw an error.

To apply the syntax action as an attribute or an operation we must first create a semantics object for the grammar.

```
var grammar = ohm.grammar(ohmGrammarFile);
var semantics.createSemantics();
```

We can then add to the semantics object with a translator object. Adding an operation or an attribute both follow the same syntax. The first argument is the name to give to the syntax action and the second argument is the object that contains the actions.

As an attribute:

```
semantics.addAttribute('trans', trans);
```

As an operation:

```
semantics.addOperation('trans', trans);
```

Ohm provides an API to help traverse through nodes and get information about them. Table 2.3 provides a description of the methods used in the JAMScript compiler.

Table 2.3: Description of Ohm methods used

Rule	Description
`child(x)`	Function that returns child number `x` attached to a node
`numChildren`	Property containing the number of children attached to a node
`ctorName`	Property containing the name of the expression that generated a node
`sourceString`	Property containing the original source string that matched with an expression to generate a node

Optionals

When dealing with rules that contain optional nodes, such as:

```
RuleA = "Hello" "World" "!"?
```

We can test check if the optional node was filled by using the numChildren property. If a node has no children then it is an optional node that was not used. If it has one or more children then it was used.

```
var actions = {
  ruleA: function(hello, world, exclamation) {
    if(exclamation.numChildren > 0) {
      return "Hello World!";
    } else {
      return "Hello World";
    }
  }
}
```

Lists

If an expression contains a list, such as:

```
RuleA = ListOf<RuleB, ",">
```

The child node that contains the list will contain three child nodes. The first node contains the first element in the list, the second element contains a list of the remaining nodes, and the third element contains a list of the separators.

```
var actions = {
  RuleA: function(first, rest, separators) {
    var output = first.translator;
    for(var i = 0; i < rest.length; i++) {
      output += seperators.child(i).translator + rest.child(i).translator;
    }
  }
}
```

2.4 Ometa Programming Language

In our initial prototype of JAMScript, the compiler was written in Ometa [9]. One of the features that was well suited to the design of JAMScript was Ometa's ability to extend multiple grammars. This allowed us to create standard C and JavaScript parsers and translators, and then build JAMScript as a language that extended both of these at once, creating in a unified language. Unfortunately, after completing a prototype of the

JAMScript language in Ometa, we found the implementation was too slow to be usable. After testing the Ometa parsing speed and looking the results that others have found in their implementations of Ometa compilers [10], we found that a typical Ometa compiler (implemented in JavaScript) can parse approximately 100 lines of code per second. This was not practical for our purposes, as in our pipeline a C program would go through the preprocessor before being sent to the compiler for parsing. As such, even simple C programs of less than a hundred lines of code can become thousands of lines long after the preprocessor includes all the necessary imported code. In our tests an input program of 50 lines could take over 30 seconds to compile, as the compiler was receiving over 3000 lines to process. We decided that this was creating unacceptably long wait times for compiling a program, and the problem would only get worse as the programs got more complicated. This led to switching away from the Ometa implementation and re-designing the compiler in Ohm, which was able to achieve must faster results.

To illustrate the performance benefits of switching to Ohm from Ometa, we ran a performance comparison between the two tools. Each was configured using a similar ECMAScript 5 parser and passed the same JavaScript input file to parse. We progressively increased the number of lines in the input file and retested the parsing time. We found that for small input files the performance difference between the two tools was not significant, but as the lines of code increased the performance of the Ometa parser began to lag behind the Ohm parser considerably.

We also compared the time to run a pretty printing translator but found that the time difference between the code generation phases in Ohm and Ometa were not significantly different. This is likely because they are performing similar tree walking actions to traverse the parse tree.

Ohm shares an overall similarity with OMeta but there are some significant design choices that differentiate the two. Like OMeta, Ohm is a PEG parser generator. Unlike in Ohm, semantic actions are not allowed in the grammar files. This creates a separation of the pure grammar files and the semantic actions. As Ohm (like OMeta) can be implemented in different languages, this allows the grammar files to be compatible with all implementations. This does create some restrictions on the programmer as there are times when it may be helpful to execute code in the parsing phase, such as collecting symbol definitions. We had also used the feature in Ometa to extend from multiple grammars at once which would not be possible in Ohm.

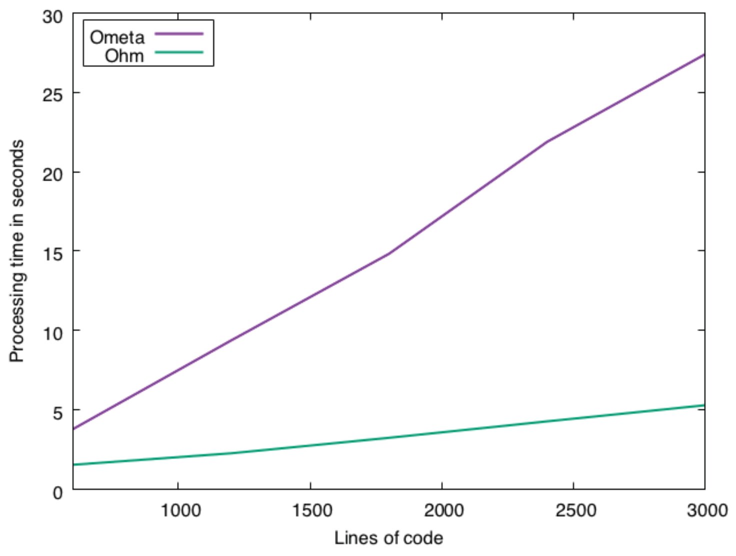

Fig. 2.1 Performance comparison of Ohm and Ometa

Equivalent grammar sample in Ohm and Ometa

Ohm

```
L {
 number   = digit+
 AddExpr  = AddExpr '+' MulExpr   -> Add
          | AddExpr '-' MulExpr   -> Subtract
          | MulExpr
 MulExpr  = MulExpr '*' PrimExpr -> Multiply
          | MulExpr '/' PrimExpr -> Divide
          | PrimExpr
 PrimExpr = '(' Expr ')'          -> Paren
          | number
 Expr     = addExpr
}
```

Ometa

```
ometa L {
 number   = digit+,
 addExpr  = addExpr '+' mulExpr
          | addExpr '-' mulExpr
          | mulExpr,
 mulExpr  = mulExpr '*' primExpr
          | mulExpr '/' primExpr
          | primExpr,
 primExpr = '(' expr ')'
          | number,
 expr     = addExpr
}
```

Ohm

```
Expr: function(e) {
    return e.eval();
},
AddExpr: function(e) {
    return e.eval();
},
AddExpr_Add: function(left, op, right) {
    return left.eval() + right.eval();
},
AddExpr_Subtract: function(left, op, right) {
    return left.eval() - right.eval();
},
MulExpr: function(e) {
    return e.eval();
},
MulExpr_Multiply: function(left, op, right) {
    return left.eval() * right.eval();
},
MulExpr_Divide: function(left, op, right) {
    return left.eval() / right.eval();
},
PrimExp: function(e) {
    return e.eval();
},
PrimExp_paren: function(open, exp, close) {
    return exp.eval();
},
number: function(chars) {
    return parseInt(this.sourceString, 10);
},
```

Ometa

```
ometa L {
  number = digit+,
  addExpr = addExpr '+' mulExpr
          | addExpr '-' mulExpr
          | mulExpr,
  mulExpr = mulExpr '*' primExpr
          | mulExpr '/' primExpr
          | primExpr,
  primExpr = '(' expr ')'
           | number,
  expr = addExpr
}
```

Chapter 3

Systems Architecture

3.1 Overview

In this chapter, we detail the systems architecture of the JAMScript compiler. The goal of our compiler system is to take in the user's input code files and finish with a running program. The compiler is made up of several components that work together to create the runnable JAMScript program. With the exception of the native compilers, all components of the JAMScript architecture are custom made for JAMScript. The architecture is modular and components can be swapped out with equivalent components without affecting other parts of the system. The components related to optimizing the runtime execution are not yet completed but do not affect the current implementation's ability to launch complete applications. This section does not go into detail about the separate architecture that exists for managing the deployment of a JAMScript application.

3.2 Components of the Architecture

JAMScript Program

The JAMScript architecture begins at the user created program. A program is written as two separate halves; the C file and the JavaScript file. The design was implemented this way to maximize the similarity with writing regular C and JavaScript when creating a program. The ordinary C and JavaScript code are augmented with JAMScript specific features to enable new functionality, such as activity calls.

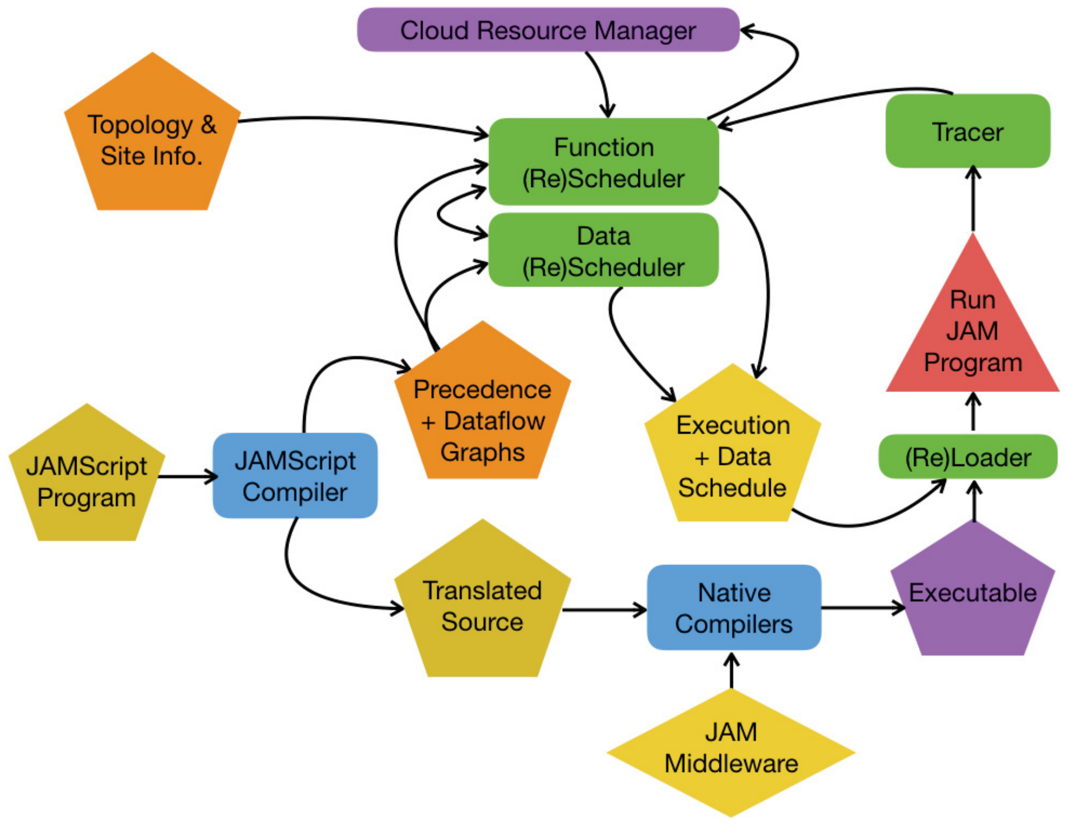

Fig. 3.1 JAMScript system architecture

JAMScript Compiler

The JAMScript compiler takes in the JAMScript program files, specified as separate JavaScript and C files. The compiler will parse through the input files to determine what code needs to be generated by looking at things such as activities used, JData declarations and jconditions. The compiler is written in JavaScript and is based on the Ohm parser generator.

Translated Source

The compiler generates translated source code for both JavaScript and C code. The translated C source code is a valid C program that can be compiled by any Clang compiler, as long as the necessary libraries are installed. The translated JavaScript code is valid JavaScript that can be run by any Node.js installation with the correct dependencies installed. This allows our code to be portable between machines when we compile and execute it later.

Native Compilers

The translated C source code is sent to Clang to be natively compiled. We make use of the complete compiler pipeline, from preprocessing to linking. Clang generates an executable file as output that will run on the local machine.

JAM Middleware

The JAM middleware is a combination of C libraries and JavaScript modules that are used in a JAMScript application to enable communication between nodes. The C libraries are used by Clang when compiling the program. The JavaScript modules are bundled with the NPM installation of JAMScript, enabling the executable to run on any machine that the JAMScript package is installed on.

Executable

The output of the JAMScript compiler is a JAMScript jxe executable. This file is a zipped folder containing the C executable file and a runnable JavaScript file. The executable can also contain additional information files, such as the call graph and jview web pages, if the program was compiled with additional options specified.

(Re)Loader

The loader is responsible for taking the JXE file and preparing it to run on the local machine. The loader can change the execution parameters of the program based on the results of the scheduler to optimize activity calls and JData performance. This can be done after the program has already launched, reloading the program with better-optimized calls based on runtime performance.

Run Jam Program

The running JAM program is a combination of the different environments and systems that the JAMScript program is running on. This can include multiple C executables running on the device, as well as JavaScript code that is running on the Node.js runtime in the device, fog or cloud.

Tracer

A tracer could be run on the JAMScript program to provide debug logging information about the code execution. The tracer would be combined with a compiler flag to output the executable with tracing utilities generated into the code.

Precedence and Data Flow Graphs

The JAMScript compiler can generate information about the layout of the JAMScript program. We generate a visual representation using JavaScript graphing tools and a machine-readable version using DOT files. The DOT file can be sent to the scheduler to optimize execution parameters.

Topology and Site Information

JAMScript is able to deploy with no topology information or additional information about the configuration setup can be supplied by the user. This will enable the scheduler to better optimize the execution of the program.

Cloud Resource Manager

A cloud resource manager is used to communicate with the function scheduler. This provides a centralized controller to keep track of the execution properties and modify them if need be.

Function (Re)Scheduler

We use a function scheduler to optimize the function call timing. This is based on static analysis using the precedence graph files generated by the compiler and dynamic analysis performed by a combination of other components of JAMScript.

Data (Re)Scheduler

The data scheduler exists to optimize reads and writes to the JData variables. The data for these variables are stored on Redis servers located on the devices, the fogs, and the cloud. The data scheduler will determine the best timing for reading and writing variables to the database. This is based on the static analysis generated by the compiler in the data-flow graphs, as well as dynamic analysis.

Execution and Data Schedule

We use the results of the function and data schedulers to modify the execution of the jam program. The execution and data schedule will be sent to the loader to change the runtime configuration of the JAM program.

Chapter 4

The JAMScript Language

JAMScript programs are written as two halves, comprised of the C portion and the JavaScript portion, that are integrated together during compilation to make a single executable. Activities are special functions that we introduce to the existing language syntaxes that allow calls to be made between different machines and languages automatically.

JAMScript programs are composed of at least two files, a C file ending with the extension `.c` and a JavaScript file ending with the extension `.js`. This separation of files allows for developers to specialize in the programming language they are most familiar with. For example, a software development team can be divided into separate C and JavaScript teams, with the only requirement for coordination between the two teams is deciding on the function signatures for JAMScript activities.

Using a combination of C and JavaScript code also allows developers more customizability in how their code is deployed. By allowing developers to write their own C code, we allow complete control over pointers and other low-level C facilities. This brings that advantages of C that may be necessary on low-powered devices, such as performance gains and a a lightweight runtime. This gives an advantage over running JavaScript directly on the device or by using ports of JavaScript for embedded devices, such as low.js[1], that only provide a subset of the API of the standard Node.js[2], and do not match the performance of C code.

[1]https://lowjs.org
[2]https://nodejs.org

4.1 C Activities

We use C activities to create functions that are callable to other nodes inside a JAMScript program. C activities are defined inside of the C input file and written similar to a regular C function.

Synchronous

Synchronous C activities are identical to regular C functions with the addition of the keyword **jsync** before the return type:

```
jsync return_type activity_name(c parameters) {
  // C code body
}
```

This creates a new synchronous C activity with the name **activity_name**. Activities can have a return type of `int`, `float`, `char`, `char*` or `void`. Any number of parameters are accepted of type `int`, `float`, `char` or `char*`. The body of the activity accepts standard C code and behaves like a normal function. When this activity is called it will block execution on the source of the call until the activity returns. Synchronous C activities that are called from a JavaScript node will return an array containing the result from each node that completed the activity.

Asynchronous

Asynchronous C activities are similar to the synchronous version, but do not have a return type:

```
jasync activity_name(c parameters) {
  // C code body
}
```

This creates a new asynchronous C activity with the name **activity_name**. The activity has no return type specified as the activity will not return a value. Since the activity is asynchronous, when the activity is called execution at the call source is not blocked and will continue without waiting for a return value. Callback activities can be used to retrieve results from an asynchronous activity.

4.2 JavaScript Activities

JavaScript activities are written in the JavaScript input file with a syntax similar to a regular JavaScript function. When writing JavaScript activities, you must declare a matching prototype for the activity on the C side. This will give the compiler information about what the return type and the parameter types of the activity which otherwise could not be determined statically at compile time for JavaScript programs.

Synchronous

JavaScript:

```
jsync function activityName(parameters) {
  // JavaScript code body
}
```

C:

```
return_type activityName(parameter_types);
```

This will create a new JavaScript synchronous activity with the name **activityName**. The body of the activity is standard JavaScript code. When this activity is called it will block execution of the calling program until the activity returns. The C prototype specifies the types for the return value and parameters.

Asynchronous

JavaScript:

```
jasync function activityName(parameters) {
  // JavaScript code body
}
```

C:

```
void activityName(parameter_types);
```

This will create a new asynchronous JavaScript activity with the name **activityName**. The activity has a return type void specified in the C prototype, as the activity will not return a value. Since the activity is asynchronous, when the activity is called execution at the call source is not blocked and will continue without waiting for a return value. To retrieve results from an asynchronous activity you must use a callback function.

4.3 Asynchronous Activity Callbacks

JAMScript supports passing asynchronous activities as arguments when calling an asynchronous activity, creating the ability to perform callback calls. This allows a remote activity to execute code on the node that made the original call to an activity.

jcallback activities can be defined as either JavaScript or C functions. A C function can be used as a callback activity if it has a return type of `void` and a single parameter of type `char *`. A JavaScript function can be used as a callback activity if it has a single parameter, which should be treated by the programmer as a String. Activities can take in a callback function as an argument by declaring a parameter with type `jcallback`. Activities declarations can have multiple jcallback parameters, which allows for the program to call different callbacks depending on the situation.

As an example, we call an asynchronous JavaScript activity from a C node and use a callback to print the result from the C node:

```c
void remoteActivity(int, jcallback);

void callbackFunc(char * result) {
  printf("%s\n", result);
}

int main() {
  remoteActivity(0, callbackFunc);
}
```

On the C side we declare the function **callbackFunc** that we will use as the callback. The function takes a single argument of type `char *` and has a return type `void`, the function signature necessary to be used for a jcallback function. We also declare the prototype for the activity on the JavaScript side, **remoteActivity**. This activity takes in an `int` and a `jcallback` as arguments. When we call **remoteActivity**, we put the name of the C function to use as callback as the argument for the `jcallback` parameter.

```
jasync function remoteActivity(value, finishedCB) {
  finishedCB("Hello " + value);
}
```

We call the callback from the JavaScript activity by using the `jcallback` variable **finishedCB**. A callback function must be called with a String as the lone argument and runs asynchronously.

4.4 Defining Shared Persistent Variables

JAMScript data storage is performed by JData, a feature for creating shared variables between multiple nodes. A JData variable can be can be declared as a logger or broadcaster. Loggers are used to save data upwards to parent nodes, while broadcasters are used to save data downwards to children nodes.

JData uses Redis [11], an in-memory data structure store, for saving variable contents and sharing variables between nodes. Each J node automatically runs a separate instance of a Redis server. Variables saved to JData are written to one or more of the most relevant Redis servers depending on what the situation requires. JData handles the reading and writing to multiple Redis servers seamlessly.

JData declarations are done by using a **jdata** block at the beginning of the JavaScript file:

```
jdata {
  int x as logger;
  char *y as broadcaster;
}
```

Multiple variables can be defined in a single **jdata** block. The C data type is specified first, followed by the variable name and whether it is a logger or a broadcaster.

JData variable declarations must be declared with a C variable type. Table 4.1 shows the C types that are currently supported in JData and the equivalent JavaScript type.

Table 4.1: JData compatible data types in C and JavaScript

C	JavaScript
int	Number
float	Number
double	Number
char	String
char *	String
struct	Object

JData also supports using structs to group together multiple variables:

```
jdata {
  int x as logger;
    struct fruit {
      int apple;
      float pear;
      struct tree {
        int leaf;
      };
    } s as logger;
}
```

A struct can contain multiple members of any supported JData type, including nested structs. A structure tag and a structure alias are required at the top level of the struct, while only the structure tag is used for nested structs. When accessed on the C side the JData struct behaves similarly to a C struct. When accessed on the JavaScript side the JData struct becomes an object, with each member of the struct as a property of the object.

4.4.1 Saving Data to Memory

JAMScript provides the ability to store data through the logger JData variable. The logger stores data from each node as a separate time series. When a node updates the value of a variable it appends the new value its time series. During declaration, loggers can be specified to store data on device, fog or cloud. Data can be written from any level at or below the one specified, but can only be read from the level specified. This means that a cloud logger can be read from device, fog or cloud but can only be read from the cloud. Table 4.2 lists the read and write access for loggers for each specifier.

Table 4.2: Logger read and write access by specifier

Specifier	Write Access	Read Access
Device	Device	Device
Fog	Device, Fog	Fog
Cloud	Device, Fog, Cloud	Cloud

Loggers are declared in a JData block in the JavaScript code. The default level for a logger is on device storage:

```
jdata {
  double x as logger;
}
```

A logger can be defined as on device, fog or cloud by adding a specifier to the declaration:

```
jdata {
  double y as logger(fog);
}
```

Writing to the logger from a C node is done by assigning to the specified logger variable as if it were an assignment to a regular variable. This can be done in any function or activity that contains C code:

```
int testFunction() {
  x = 32.0;
}
```

Writing to the logger from a J node is done by using the logger object's method. The log method takes in a single value to append to the time series:

```
function testFunction() {
  y.log(24.1);
}
```

Reading from a logger is done on the J node at the level specified when declaring the logger. The logger variable on the JavaScript side is represented as an object which provides methods to read the values. The data is stored as an array of time series', where each node writing to the logger is an element of the array. If a device had 4 C nodes writing to the logger, the JavaScript object would have an array of length 4 with each element containing a time series.

Structs

Logging to struct must be done in a single operation to ensure that all values in the struct are pushed to the time series in a single update. There are two ways to write to the logger from the C side; assigning to all logger struct members at once or using a local struct to store the values before assigning to the logger. When saving to a struct logger, all members must have a value assigned or an error will be thrown.

For example, if we had the struct logger **point** declared:

```
jdata {
  struct cords {
    int x;
    float y;
  } point as logger;
}
```

One way to save to this logger would be to use tagged structure initialization to assign all members of the struct at once:

```
point = {
  .x: 5,
  .y: 2.0
};
```

Another way to save to the logger is to use a struct variable to temporarily store the values before assigning them to the logger. Declaring a struct in JData will generate a C struct definition with the same members that can be used to declare a local struct:

```
struct cords localCord;
localCord.x = 3;
localCord.y = 4.0;
point = localCord;
```

Logging to the struct from the JavaScript side is done by creating an object with all properties of the JData struct:

```
point.log({x: 5, y: 1.0});
```

Reading from the logger struct is done by accessing the properties of the object on the JavaScript side:

```
console.log(point[0].lastValue().x);
```

Methods

The methods available through the JavaScript interface when reading from a logger's time series are described in Table 4.3.

Table 4.3: Description of logger methods

Method	Description
size	Returns the number of elements in the time series.
data	Returns an array containing all data pairs of type (value, timestamp) for the time series.
values	Returns an array containing all values for the time series.
lastData	Returns the last data pair (value, timestamp) in the time series.
lastValue	Return the last value in the time series.
dataAfter(Date)	Returns an array containing all data pairs of type (value, timestamp) in the time series that occur after a specified Date object.

valuesAfter(Date)	Returns an array containing all values in the time series that occur after a specified Date object.
dataBetween(Date, Date)	Returns an array containing all data pairs of type (value, timestamp) in the time series that occur between two specified Date objects.
valuesBetween(Date, Date)	Returns an array containing all values in the time series that occur between two specified Date objects.

When loggers are used on multiple nodes each node has a separate data stream. Each data stream is structured as a time series; when data is logged it is appended to the end of the series along with the timestamp of the insertion. An array of time series' is used to store each node writing to the logger as a separate element. Loggers on different nodes can push data at different rates, so each time series can have a different length.

In Table 4.4 an example application using a logger is running on 3 C nodes. The logger will have an array of 3 elements, each containing their own time series. Using the `size()` method on the logger object will return the number of elements in the array; the number of nodes writing to that logger.

Table 4.4: Example of nodes with varying logger contents

Node 1	Node 2	Node 3
(v11, t11)	(v12, t12)	(v13, t13)
(v21, t21)	(v22, t22)	(v23, t23)
(v31, t31)		(v33, t33)
		(v43, t43)

Individual data stream's in the logger can be accessed by referring to their element in the logger array:

```
jdata {
  double x as logger;
}

x[0].lastValue();
```

4.4.2 Reading Data from Memory

The JData broadcaster variable provides a mechanism for pushing data from a parent node to its children using a shared memory store. The data is stored at the level specified when the broadcaster is declared. Any node below the level specified can read values from the broadcaster but only nodes at the level specified can write a value. Table refbroadcaster-Access lists the read and write access for broadcasters for each specifier. While the logger contains multiple streams, each containing many values, the broadcaster only contains a single value at a time.

Table 4.5: Read and write access by specifier

Specifier	Write Access	Read Access
Device	Device	Device
Fog	Fog	Device, Fog
Cloud	Cloud	Device, Fog, Cloud

A broadcaster is defined with the broadcaster keyword in a JData declaration:

```
jdata {
  double x as broadcaster;
}
```

The broadcaster can be written to from fog or cloud nodes. Conditionals can be used to control where the data originates. For example, the execution of an activity that writes a value to a broadcaster may be restricted only to fog nodes.

Below we show broadcaster data that can only originate from a fog node:

```
jdata {
  double x as broadcaster;
}
jcond {
  fogonly: sys.type == "fog";
}
jasync {fogonly} function fname() {
  x = 2.2;
}
```

When broadcaster data is read, the latest value is always used. A function referencing a broadcaster does not execute until the node on which the program runs has received the latest value of the broadcaster. A condition referencing a broadcaster is always evaluated using the latest broadcaster value. In order to ensure producer-consumer synchronization, the JAM runtime performs versioning of the broadcaster values. Example:

```
jdata {
  double pe as broadcaster;
}
jcond {
  pickpe: pe < sys.rank; // Evaluated using the latest broadcaster value
}
```

```
jasync {pickpe} function fname() {
  double y = pe; // Assigns the latest broadcaster value
}
```

Reading from the broadcaster on a C node is done by assigning the value of the broadcaster variable to another variable:

```
void testFunction() {
  int localStore;
  localStore = x;
}
```

Structs

Broadcasting with a struct is similar to logging with a struct. All members of the struct must have a value specified in the object when broadcasting a new value.

```
jdata {
    struct time{
        int hour;
        int minute;
    } clock as broadcaster;
}
```

Writing to the broadcaster the object must be in a single statement with all fields specified:

```
clock.broadcast({hour: 3, minute: 10});
```

Reading the struct from the C side is done by assigning the value of the broadcaster to a local struct. The definition for the local struct is provided by the compiler and can be used to declare struct variables.

```
struct time localStruct;
localStruct = clock;
printf("%i:%i\n", localStruct.hour, localStruct.minute);
```

4.4.3 Conditional Execution Constructs

JAMScript provides the ability to limit which nodes an activity executes on by using the jconditional declaration. Conditionals can be used to limit execution of an activity to only certain levels, such as on the device, fog or cloud. Conditionals can also restrict execution based on the attributes of a node or the current value of JData variables.

Conditional Definitions

Conditionals are defined by using the jcond declaration. A jcond block can contain multiple named conditional expressions. If the jcond is unnamed then it belongs to the global namespace by default:

```
jcond {
  ruleA: 1 < 2;
  ruleB: 4 > 2;
}
```

jcond blocks can also be named, creating a namespace for the conditionals contained inside:

```
jcond name {
  ruleA: 1 < 2;
  ruleB: 4 > 2;
}
```

Conditional rules support all JavaScript non-strict comparison operators and logical operators:

```
jcond {
  ruleA: 1 < 2 && 3 == 3 || 4 >= 2;
}
```

A conditional rule can also optionally specify a function that will run if the rule evaluates to false:

```
jcond {
  ruleA: 1 < 2, executeMe;
}

function executeMe() {
  // Code to execute
}
```

JData logger variables can be used in conditional rules. When the condition is executed, the latest value saved to the logger will be retrieved and used in the comparison:

```
jdata {
  int temp as logger;
}

jcond {
  cold: temp < 0;
}
```

A JData logger is comprised of many time series', one for each node writing to the logger. When a rule referencing a logger variable is used on a JavaScript activity, the latest value from the time series that was last updated is used. When a rule referencing a logger variable is used on the C side, the latest value from the current node's time series is used.

The **sys** global context object contains information about the system state for each node. The **type** property of the sys object will always contain the level that the current node is running at: device, fog or cloud. By default, a JAMScript activity will execute at all levels of the JAMScript hierarchy. When a call for a function is initiated by a C node, then the device, fog and cloud will all receive the call for execution. By using the type information, we are able to create rules that limit the execution of an activity to specified levels:

```
jcond {
  deviceOnly: sys.type == "device";
  fogOnly: sys.type == "fog";
  cloudOnly: sys.type == "cloud";
  fogOrCloud: sys.type == "fog" || sys.type == "cloud";
}
```

Using Conditionals

Conditionals can be assigned to activities by using a tag in the activity declaration:

```
jasync { limiter } function javaScriptTest() {
  // Code to execute
}
```

Conditionals are language agnostic, and the same rules can be used on C activities:

```
jasync { limiter } void cTest() {
  // Code to execute
}
```

Rules inside of namespaces can be used by specifying the namespace:

```
jasync { namespace.rule } function test() {
  // Code to execute
}
```

Logical operators can be used to join together multiple rules:

```
jasync { ruleA || ruleB && !ruleC } function test() {
  // Code to execute
}
```

In the following examples we use the same logical operators but at different points to achieve the same functionality. Our goal is to create an activity that will only run on the fog or the cloud. One way to achieve is this to create a rule that states the entire condition. Putting more functionality into a single rule can be useful to show that certain condition tests should always be done together:

```
jcond {
  fogOrCloud: sys.type == "fog" || sys.type == "cloud";
}

jasync { fogOrCloud } function test() {
  // Code to execute
}
```

Alternatively, we can achieve the same functionality by writing separate rules and joining them together at the activity declaration. This allows writing more modular rules that can be used separately depending on the situation:

```
jcond {
  fogOnly: sys.type == "fog";
  cloudOnly: sys.type == "cloud";
}

jasync { fogOnly || cloudOnly } function test() {
  // Code to execute
}
```

4.4.4 Conditionals over Data

Conditionals can contain JData variables to alter which activities will execute during run-time. This can be data written by the devices or those created through computations, such as flows. When using time series data in a comparison average values are computed by default. The user may alternatively use the `max`, `min`, `sum` or `std` of the time series values. The average, minimum and max values are computed over a moving window of values of the time series. When multiple streams, representing multiple devices, are present in a variable then the window spans all available streams.

```
jcond {
  temp: max(x) < 22.5;
}
```

The condition **temp** will be true if the maximum temperature value reported by all devices is less than 22.5.

Conditionals can also use JData structs in comparisons by using the aggregate functions on member fields. In the following example we take the average of a member of the struct:

```
jdata {
  struct temp_reading {
    int tempvalue;
    int xcoord;
    int ycoord;
  } readings as logger;
}

jcond {
  low_temp: avg(readings.tempvalue) < 22.5;
}
```

The logger pushes data from the devices towards the cloud through the fogs, while the broadcaster pushes data from the cloud to the devices. This means that the logger can have many sources while the broadcaster can only have a single source if it is rooted at the cloud or many, but few, sources if rooted at the fogs. In all cases, a broadcaster has a single stream.

4.4.5 Data Visualization and Controls

JAMScript provides a method for displaying and editing JData variables live through a web browser through with the jview feature. When a jview is declared, a web application is generated with graphical elements that interact with the data stored in JData. A user can specify what variables they want to use and how they want the pages configured when writing the JAMScript application. A website is generated at compile time that can run on a provided HTTP server. A socket.io[3] WebSocket connection from the running JAMScript application to the HTTP server is used to relay changes in the JData variables. The web interface is written in ReactJS[4] and uses echarts[5] [12] for displaying graphs.

The jview declaration is written on the JavaScript side of a JAMScript application. A jview declaration is comprised of pages, each with their own unique URL. Each page is made up of a combination of graphical elements, which can be displays or controllers.

[3]https://socket.io
[4]https://reactjs.org
[5]https://ecomfe.github.io/echarts-doc/public/en/index.html

Displays are used to graph the contents of a JData variable, while controls are used to change the contents of a JData variable.

Displays are used to visualize the data from JData logger variables, pulling new data automatically. The currently supported display types are `graph`, `scatter` and `stackedgraph`. New values are pulled from the JData variable according to a refresh rate that is optionally set by the user, with a default rate of 500ms. At each refresh, the latest value in the variable is retrieved and appended to the graph. A title can be specified for labelling the graph in the web page. The JData variable to retrieve data from is specified using the source tag.

Controls are used to modify the value of a JData broadcaster variable. When the value in a control element changes, the updated value is sent to the JData variable. A control's input type can be set to `slider`, `button` or `terminal`. Each control must have a JData variable assigned in the sink option, specifying which variable receives data from the element. The `slider` controller allows the user to move a slider, automatically sending the selected value to the sink variable. The `button` controller creates a clickable button that toggles the current state between true and false. The sink variable is used to store the value received from the button, alternating in value as either a one or a zero, corresponding to true and false states from the button. The `terminal` type provides the ability to type text and pass it to the sink variable. A title can be specified for labelling a control element in the web page.

Below we show an example of declaring a jview. We begin with declaring the following JData variables that will be used as sources and sinks for jview elements:

```
jdata {
    int posDisplay as logger;
    float tempDisplay as logger;
    int slideControl as broadcaster;
    int buttonControl as broadcaster;
}
```

We can then use these JData variables inside of a jview declaration. We begin by declaring two pages with the ids **page1** and **page2**. page1 has three display elements; one display and two controllers. Using the name option changes the title for the page to **Thermostat**. **elem1** is a display that will read the latest value from the **tempDisplay** logger and show it in a line graph format with a refresh rate of 200ms. **elem2** is a slider controller, when the user moves the slider the selected value will be sent to the JData broadcaster **slideControl**.

elem3 is a button controller, clicking on the button will send a value to **buttonControl** of either zero or one depending on the state of the button. page2 contains just one display element and since the page has no name option specified the id page2 will be used as its name. **elem4** is a scatter plot display that will read from the **posDisplay** logger. As the refresh rate is not specified for this element, it will default to checking the data in the variable every 500ms.

```
jview {
    page1 as page {
      name: 'Thermostat';
        elem1 is display {
            type: graph;
            title: 'Temperature Display';
            source: tempDisplay;
            refresh: 200;
        }
        elem2 is controller {
            type: slider;
            title: 'Set Temperature';
            sink: slideControl;
        }
        elem3 is controller {
            type: button;
            title: 'Power';
            sink: buttonControl;
        }
    }
    page2 as page {
        elem4 is display {
            type: scatter;
            title: 'Position Display';
            source: posDisplay;
        }
    }
}
```

4.4.6 Data Stream Filtering and Transformations

The JAMScript Flow is a data abstraction that allows several operations on JData data streams, files and data structures. Using Flows allow large collections of data to be processed efficiently.

The Flow uses filters and transformations to create a declarable programming style which allows easy creation of new Flows and to modify existing ones. A Flow allows the programmer to operate on data in a way similar to SQL commands and queries. Flow operations can either be methods, transformations or actions. The method and transformation operations are used to produce new Flows, which can be used in future operations. Action operations are used to produce a final value and do not produce a Flow.

Sample Flow example in JAMScript

```
jdata {
  double x as logger;
  q as flow with <flowFunc> of x;
}

function flowFunc(x) {
  return Flow.from(x[0])         // Create a Flow from a data stream
    .select((data) => data.log) // Select a part of the data set
    .where((data) => data > 25) // Filter for values greater than 25
    .limit(100);                // Limit to the first 100 elements
}
```

Every Flow definition must have a JavaScript function associated with it that defines the operations that will be used to transform the data. A Flow is lazily computed and will only be processed when an action is called on it.

As an example:

```
let firstValue = q.findFirst();
```

Using the **findFirst** method causes the Flow to be processed and to return the first value found. Processing stops immediately after a result that satisfies the constraint is found, saving processing time.

Flow Creation

A Flow can be created from several JavaScript and JAMScript data structures including: Array, Set, Map, Object, FileSystem, Generator, JAMDatasource and JAMDatastream. The last three can be used to produce an infinite stream of data.

As an example, we can create a Flow from a regular JavaScript array:

```
var array = [1, 0, 5, 13, -1];
var flow = Flow.from(array);
```

Flow Types

There are four different Flow types: IteratorFlow, OutFlow, InFlow and Flow (the default). A Flow's type decides what operations can be performed on it.

- Flow: This is the default Flow that has all the basic operations for data processing.
- IteratorFlow: This Flow is used early on in a Flow chain. When the **from** method is called, an IteratorFlow is created. This Flow extends the default Flow and provides additional operations.
- OutFlow: This Flow is responsible for processing and sending data to other applications.
- InFlow: This Flow is responsible for receiving data from other applications.

Flow Chain Pipelining

A Flow chain is a linked data structure of different Flow objects. Every Flow is aware of the previous Flow and the next Flow in the chain. A Flow chain is created when a Flow method is called on a Flow object.

For example:

```
var flow = Flow.from(array).skip(2).where((num) => num % 2 == 0);
```

In the example above, there are three Flow objects in the Flow chain. When an action is called on the final Flow object in the chain, the data is piped through the Flow chain until it gets to the last Flow, where the action is computed.

Flow Push and Pull Models

Flow provides two models for data pipelining; the push model and the pull model. The pull model is used to request that data be piped from an IteratorFlow to the rest of the Flow chain. The data is generated from the IteratorFlow on request and sent through the chain. This model is used by Flow actions to do a final computation on the dataset. In the push model, data is automatically piped through the Flow chain. The push model is used in Flow Streaming.

Flow Streaming

For continuous streams of data, Flow provides a data push model that can continuously pipe data through the Flow chain. This can be useful if the computed data will be sent to another application for further processing. A Flow pushes processed data to the next Flow in the chain or if the Flow is the last in the chain to a terminal function. The terminal function for a Flow can be set using the **setTerminalFunction** method. Flow streaming can be used when the Flow is created from a JAMDatastream, a JAMDatasource or a function that generates continuous data such as a JavaScript Generator.

IteratorFlow

The IteratorFlow is a Flow that creates a unified means of retrieving data from different data structures. The IteratorFlow turns the data passed to it using the **from** method into a JavaScript Iterable by wrapping the data with an iterator implementation that allows retrieving data by calling the **next** method on the iterator handle.

The IteratorFlow is the root of the Flow chain and can be accessed from any Flow in the chain by using the property **rootFlow**:

```
var flow = Flow.from(array).skip(2).where((num) => num % 2 == 0);
var iteratorFlow = flow.rootFlow;
```

For data streaming in Flow, the IteratorFlow listens for changes on the JData variable, if changes are detected then the new value is retrieved. The retrieved data is then pushed through the Flow chain until it gets to an OutFlow or the terminal function of the last Flow object in the chain. To start data streaming in Flow, the **startPush** method needs to be called on an IteratorFlow object. To stop the streaming, the **stopPush** method can be called on the IteratorFlow object. When the stopPush method is called the IteratorFlow stops listening for incoming data on the connected streams.

OutFlow

OutFlow is a specialized Flow that is built for the purpose of sending processed data to external applications. When an OutFlow is created, a Flow object is supplied as an argument which the OutFlow links to in order to receive pushed data. OutFlow logs all received data to Redis.

In the example below, we create a new Outflow **p** that takes the output of Flow **q** and makes it available to other applications:

```
jdata{
  double x as logger;
  q as flow with flowFunc of x;
  p as outflow of q;
}

var q = flowFunc(Flow.from(x));
```

In JavaScript, `p as outflow of q;` becomes:

```
var p = new OutFlow("p", q).start();
```

The OutFlow method **start** calls the **startPush** method in the IteratorFlow, the first flow in the chain, and informs the IteratorFlow to start listening for push data from the data source. This data is continuously pushed and may reach the OutFlow if the constraints of each Flow object in the chain allow the data to pass through. Once the data arrives at the OutFlow it is sent to the Redis server.

To stop listening to data streams changes, call the OutFlow **stop** method on the object handle, which will in turn call the stopPush method on the IteratorFlow. For example, to stop our previously declared OutFlow **p**:

```
p.stop();
```

InFlow

The InFlow is a specialized Flow that is responsible for retrieving data from external applications and making it available for use inside the current application. The retrieved data can be processed using Flows before being used. The InFlow listens for new data from the OutFlow of other applications and retrieves the data using Redis.

As an example:

```
jdata {
  r as inflow of app://app1.p;
}
```

We can use the data received in a local function immediately:

```
r.setTerminalFunction(doReceiveData);
```

We can also use the data in a Flow chain:

```
var flow = r.select((input) => input.temp).where((t) => t <= 37);
```

Flow Methods

Flow methods are data transformations on a Flow that produce another Flow. Each Flow maintains a call tree that keeps a link to the Flow operation before it. Flow methods are lazily computed and data is continuously piped to the next Flow for further processing. This can reduce the execution time because some operations can be handled together. The currently supported methods are listed in Table 4.6.

Table 4.6: Description of Flow methods

Method	Description
limit(Number)	Limits the number of results obtained after the previous operation. This is a filter operation
skip(Number)	Ignores the first specified number of results found after the previous operation. This is a filter operation.
select	Similar to a map function in MapReduce operations, selects one or more parts of data from a given dataset.
selectExpand(function)	Maps one input to many outputs, as generated by the function. The collection generated by this function must be supported by the Flow **from** method.
selectFlatten	Similar to selectExpand, but doesn't take a function as an argument. selectFlatten assumes that the input from the pipe is a collection that is supported by the Flow **from** method.
where	Performs a filtering operation on the data to match a constraint.
orderBy	Performs a sorting operation on the data based on a given function. Flow has internal operations to sort using ascending or descending order of a given key from the data.
distinct	Returns a Flow with unique elements (a set) using the JavaScript strictly equals operator.
groupBy	Returns a new Flow containing several datasets grouped by a specified key. The value in the key defines the grouping and the value comparison is done using the JavaScript strictly equals operator. Another definition with a given function can cause this method to act as a partitioner, as in MapReduce, thus the groupings could be via a range as offered by the provided function.

merge(data)	This method is only available to an object of IteratorFlow and is used to merge a supported data structure.
range(from, to)	Combines the limit and skip methods, creating a bound for the data to be used for further processing.
discretize(count, length)	This method is only available to the IteratorFlow and allows processing data streams into windows. count is the number of data streams to focus on in a window. length can either be a Number or a function that checks if it's the end of the window.
setTerminalFunction(func)	Specifies a function to send push data to. This method can only be used when on a Flow object that is the last in a Flow chain.

Flow Actions

Flow actions are operations that produce results that are not Flows. When an action is called on a Flow, the Flow engine begins operating on the data and pipes each produced data to the next layer until the condition for the action is met. The currently supported actions are listed in Table 4.7.

Table 4.7: Description of Flow actions

Action	Description
count()	Returns the total number of datasets left after the last Flow method.
collect(function)	Returns the Flow data as a specified data collection. The input argument function can be one of three Flow internal functions: `toSet()`, `toArray()` or `toMap()`. toSet returns a distinct dataset, toArray returns all the data remaining after the last Flow method as an array. toMap returns a Map; this can only be used when the last Flow method operation was groupBy.

foreach(function)	Sends the remaining data from the last Flow in the chain to the specified function. This is useful when operating on every data of a Flow outside the context of Flows.
anyMatch(function)	Returns a Boolean with the result of checking if any remaining data in a Flow matches the definition in the user defined function.
allMatch(function)	Similar to anyMatch, checks that all the remaining data matches the condition defined in the specified function.
noneMatch(function)	The inverse of allMatch, checks that none of the remaining data matches the specified function's condition.
findFirst()	Returns the first data found in a Flow.
findLast()	Returns the last data found in a Flow.
findAny()	Returns any data from the Flow. Returns the same result as findFirst but can provide addition functionality in a parallel computing scenario.
reduce(initial, function)	Used to reduce a Flow to a single value. Takes in an initial value for the reduce operation and a function that defines how the reduction should be performed.
average()	Produces an average of the items in a Flow. As this is restricted to the Number type data, the previous Flow operation must produce Numbers.

Flow actions are not optimal when working with continuous streams of data because the data is not finite when the operations are carried out. Using the discretize Flow method to separate the data into windows is recommended before performing actions on continuous streams.

Chapter 5

Design of the Compiler

The implementation of JAMScript required a custom compiler to be written, which was done using the Ohm parser generator. Using Ohm allowed us to quickly modify the language design of JAMScript as we built the language and test how these changes would affect the usage of the language. One of the advantages of implementing JAMScript as a new programming language is the ability to perform analysis on the code of variable and function usage and perform optimizations and scheduling, that will be more beneficial in the future for JAMScript.

The compiler takes in C and JavaScript files that are augmented with JAMScript structures and generates a jxe executable file that can then be run by the user. The C code is generated as standard C99 code that can then be compiled and run on any machine with an installation of Clang[1] and Node.js. The outputted JavaScript code will run on any machine with an installation of Node.js.

The first step of JAMScript compilation is reading in a JAMScript C file and a JAMScript JavaScript file. Both files must be present before the compiler will proceed.

5.1 C Preprocessing

The JAMScript compiler begins by running a preprocessor on the C code. We use a preprocessor at this stage so that when we parse the C code we can perform a more thorough code analysis. For example, we can see where the various variables and functions

[1]https://clang.llvm.org

were declared. We run the C code through the Clang compiler's preprocessor and save the result. We chose to use the Clang preprocessor as we are already using Clang to compile the final C code to an executable, so we share the use of that dependency to know that we have the preprocessor already installed on a given machine.

We found that most publicly available C preprocessors make use of C compiler extensions that are not defined in the ANSI/ISO C specifications. The extensions used would vary from machine to machine, even when using the same preprocessor and input code. We decided that we could not create a complete parser specification that could handle all the C extensions. It became too complicated to account for all the unique cases we encountered when using C preprocessors, and we couldn't guarantee that the JAMScript compiler would be compatible with untested machines that could introduce new extensions we had not seen before.

To solve this problem, we made use of fake C headers. This method works by including mostly empty C files with filenames that match with the filenames of the headers in the C standard library. The only information declared in these files is generic typedefs for most common variable types. The analysis that we wish to perform is not affected as it is focused on the user-defined declarations and not those in the standard library. When the user's input code is first sent to the compiler, we save the original include statements and then run the preprocessor on the code. When the preprocessor outputs the code, the code generated can still compile but will not produce a usable executable. We then make the code usable by removing all the code that was generated from the include statements and replacing it with the original include statements. To remove the code generated by the fake headers we use a marker inserted after the include statements in the original source code. As the code generated by a preprocessor for an include statement will be inserted at the beginning of the output code, we can safely delete all code before the marker without removing any of the user's code. As our analysis is complete at this point, we can use Clang's built-in preprocessor to parse the include statements correctly and then send it to the compiler.

5.2 Ohm Compiler

The next stage of the compilation process is to run the code through our Ohm based compiler. The JAMScript Ohm compiler is divided into two sub-compilers; a JavaScript compiler and a C compiler. The input for the C compiler is the preprocessed C file with fake headers, and the input for the JavaScript compiler is the unmodified JavaScript file. Each of the compilers is divided into two portions; a parser and a translator. Ohm does not allow custom logic inserted into the parsing phase, only building the parse tree is allowed. Although we are walking through the code twice, once when building the tree and a second time walking the tree to generate output code, because of this restriction we can only add custom logic when walking the tree. This creates the same restrictions as if we were using a single walk compiler, therefore, the order that we perform our compilation matters between the two languages. The JavaScript compiler is run first because most of the JAMScript structures, such as JData and jconditional are defined on the JavaScript side.

Ohm Parser

The C parser is based on a modified C95 specification. We support all C95 features as well as some popular C99 features that we believed most users are accustomed to, such as double slash single line comments and mixed declarations and statements. We created an Ohm grammar file **jamc** that extends from our standard C grammar. This adds the JAMScript additions that are allowed in the C files, such as activity definitions and jcallbacks.

The JavaScript parser is based on the ECMAScript parser that is distributed with Ohm. We use an ECMAScript 6 parser that extends from an ECMAScript 5 parser. The only ECMAScript 6 addition that we currently support is arrow functions. We decided that most major JavaScript functionality could still be achieved by using ECMAScript 5 features. Similar to the C side, we created a grammar file **jamjs** that extends from the base es6 grammar. This grammar has more additions to the language than the C side, as the JavaScript files contain JAMScript features such as JData declarations and jconditional declarations.

Ohm Translator

The respective output trees from the two parsers are then sent to separate JavaScript and C translators. The translators walk through the trees and generate the appropriate output code. Each translator generates separate C and JavaScript code, which are later merged together into one output file per language. We use a shared symbol table between the C and JavaScript compilers that allows us to look up JAMScript symbols that could be shared between the different languages, such as JData variables and activities. Local variables and functions are not stored in the symbol table, so they are not considered when moving between the JavaScript and C compilers.

Many JAMScript features have a syntax similar to regular C code, therefore we have to check the user's input to see if they are trying to use a JAMScript feature and replace the code with the appropriate function calls. For example, assigning to a JData logger is similar to assigning to a regular C variable. In order to output the correct code, when we see an assignment to a variable we check to see if the variable name is in the symbol table and if it has a JData type. If it does, then we output function calls to save the value to the JData variable. If the variable name is not in the symbol table as a JData variable, then we can output the original input code.

5.3 Generating Call Graphs

JAMScript can generate a unified call graph that contains information about all function and activity calls throughout a JAMScript program. The call graph stores information about the source language of the call, the arguments, whether it is an activity or function call, and the target language of the call. The call graph is used internally when compiling to check the activity matrix, to see if a call is permissible when an activity calls another activity, and is also used for external analysis. In order to make the information to be the most relevant, we only put functions that the user defined into the call graph. We do not include functions that were included by the preprocessor in the C code or calls to global functions and external functions in JavaScript. The call graph is generated by the compiler into two separate file types that contain the same information. The first file type is a DOT graph description file. This file type is a standard for graphs and can be used to represent directed graphs, such as call graphs. We output the DOT file for future use, as it may be beneficial for further scheduling and program analysis. We also generate an HTML file for a visual representation of the call graph. This creates a way to easily read and understand the call graph without having to run a graphing program on the DOT file. We decided to customize the display of the file and generate a static website that presents the call graph in a controllable and easy to understand way. The webpage uses the Cytoscape.js [13] JavaScript visualization library with the dagre[2] directed graph layout engine.

[2]https://github.com/cytoscape/cytoscape.js-dagre

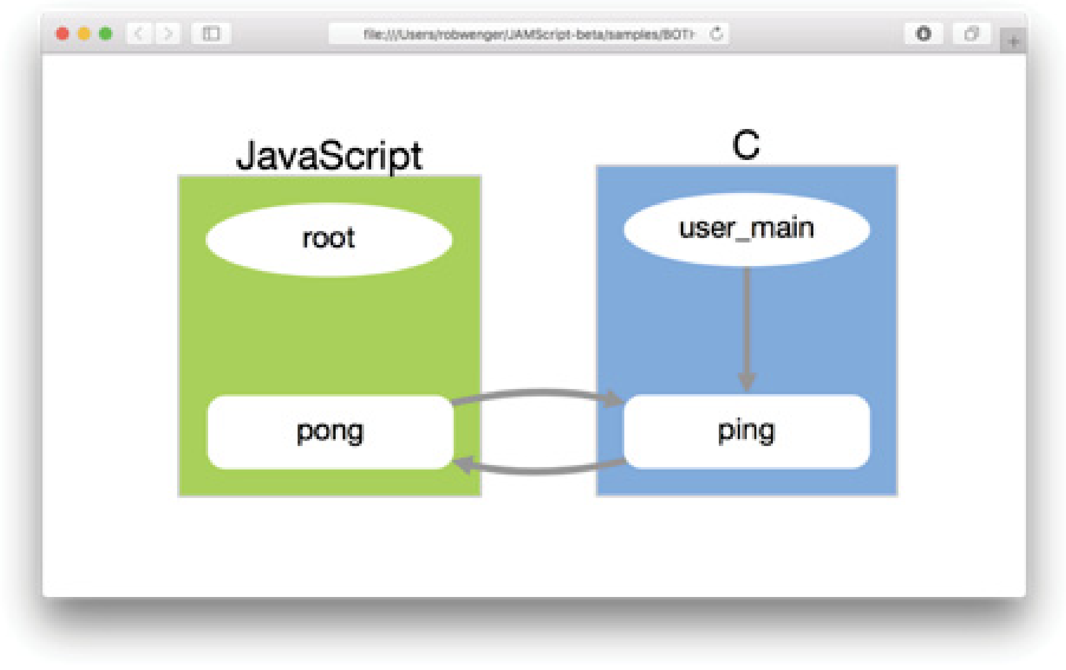

Fig. 5.1 Display of a callgraph.html file

In Fig. 5.1 we can see how the callgraph.html file is shown in a web browser. Functions and activities that are written in JavaScript are placed in the area with a green background, while those written in C are placed in an area with a blue background. Normal functions are displayed in rectangles and JAMScript activities are displayed in rounded rectangles. Arrows represent calls, and by clicking on an arrow a popup menu will display with a list containing the arguments that were used each time that call appears in the code. Calls made at the global scope in JavaScript (outside of any function) are labelled as originating from root, while those declared in the main function in C are labelled as originating from user_main. In this example, we have a sample program that starts with the user_main function calling the activity ping on the C side. This starts a loop of ping calling the activity pong on the JavaScript side, which in turn calls ping on the C side. We can also see from this chart that there are no functions calls from the JavaScript side originating from the root, the JavaScript side is waiting for the C side to start before it makes any calls.

5.4 JavaScript Type Checking

One of design goals of JAMScript is to keep the JavaScript portion of the user's code as close to original JavaScript as possible, such as not forcing type information on all variables, in order to minimize the amount of learning necessary for ordinary JavaScript developers. This precluded the use of a typed JavaScript superset language, such as TypeScript, on the JavaScript code.

In order to bring some of the protections that type checking offers to the JavaScript code of JAMScript, we integrated the Flow[3] static type checking utility into the JAMScript compiler. While JavaScript is a dynamically typed language, and C has much stricter type restrictions than the types that exist in JavaScript, we can still perform some type checking with the help of Flow. For example, if an activity takes has a parameter of type int, we can check to make sure that the user is not passing in a String or another non-number type. Flow is designed around the idea of using limited available typing information to *flow* down the program, inferring types along the way. We decided that this would be a good match for JAMScript, as the only type information that we receive is from activity parameter types and return types.

When we annotate the JavaScript code with Flow type, the code produced is not directly runnable by Node without stripping out the annotations by using a tool such as Babel. We decided the simplest way to have type checking and a runnable program was to generate two versions of the JavaScript code; one that is plain JavaScript that can be run by Node and a second with Flow type annotations that we use with the type checker. When the compiler generates the JavaScript output, the regular JavaScript code in the JAMScript program is placed into both files identically. The code that is generated for activities is created as two versions; one version that is plain JavaScript code and one that has Flow type annotations on the activity parameters and return type. The Flow type checker is then ran on the annotated file, which is then discarded. The non-annotated JavaScript file is the only JavaScript file that is placed inside the jxe file to be later launched by jamrun.

[3]https://flow.org

We can set the type of a variable in Flow by using a special syntax of annotating the code with types for variables. This prevents the user from assigning anything but a value of that type to the variable:

```
var x:number;  // Setting the type of variable x to Number;
x = 4;
x = 12.3;
x = "Hello";   // Incompatible type error
```

We can use Flow to set the type of the parameters and the return value of functions. As well, Flow can infer, based on the limited type information it has, assumptions about what is permissible and what is not:

```
function doubler(input:number): number {
  return input * 2;
}

doubler(3);
doubler("Text");  // Incompatible parameter type error
```

We can combine the information that we are able to ascertain from an activities' C prototype declaration to be able to provide some Flow annotations for our code.

For example, if we had the following C prototype:

```
char* evenOrOdd(int);
```

And a JavaScript activity:

```
jsync function evenOrOdd(x) {
  if (x % 2 == 1) {
    return "Odd";
  } else {
    return "Even";
  }
}
```

We can then annotate the JavaScript code with approximations of the corresponding C types:

```
function evenOrOdd(x: number): String {
  if (x % 2 == 1) {
    return "Odd";
  } else {
    return "Even";
  }
}
```

By annotating the activity code, we can bring some static analysis type checking into the JavaScript code. In this example, this would make sure that the user always treats **x** like a number and that there is a return statement on all branches with the return type of String.

5.5 Generated Files

When the Ohm portion of the compiler is done generating code, we perform the final two tasks at the same time. As they are both longer tasks, we use promises to spawn two shell processes that execute the tasks simultaneously. The first task runs the Flow type checker on the annotated JavaScript code. This begins with a check to see if the user has Flow installed, if they do not then we skip the type checking phase and pass the task. If the user does have Flow installed then we run the type checker and display any errors that are found. The second task runs the Clang compiler on the generated C code. When both of these tasks complete successfully, then the compiler has finished and compilation succeeded. If either does not complete successfully then we end compilation with an error.

The JAMScript compiler generates a jxe executable; a zip file containing the compiled C code and the runnable JavaScript code. Depending on the options specified when calling the compiler, additional files can also be generated. Table **??** describes all possible output files that the compiler can generate.

Table 5.1 Files generated by the JAMScript compiler

File	Description
.jxe File	A single executable for the entire program that can be launched using jamrun.
jamout.c	The complete C program file for the client devices. This file is a standard C file that can be compiled on any machine with the necessary libraries installed.
a.out	The file jamout.c after being compiled using the Clang compiler.
jamout.js	The complete JavaScript program, this file be run on any machine with Node.js and the JAMScript npm package installed.
callgraph.dot	The call graph in a DOT graph language file. This file is used for further analysis.
callgraph.html	The call graph as an interactive HTML webpage. This file is used as an easy way to visualize the call graph.

5.6 Project Directory Layout

When jamc is called, the JAMScript compiler launches from the index.js file in the root project directory. The index.js file imports files from the folder lib/ohm, which contains every other file used for the JAMScript compiler. The C and ecmascript folders contain compilers for their respective languages and could operate without the other as a standalone compiler. The jamscript folder contains the JAMScript compiler which extends from the other two compilers. A brief description of each file used in the JAMScript compiler is provided in Table 5.2.

Table 5.2: Description of JAMScript compiler files

File	Description
index.js	Starting file of the JAMScript compiler. Handles input argument processing, calling the compiler, writing output files, calling the local Clang C compiler and running the Flow type checker.

Table 5.2: Description of JAMScript compiler files

File	Description
Folder lib/ohm/C	
c	The translator for an implementation of a standard C pretty printer.
c.ohm	The Ohm grammar for a standard C parser
pretty.js	The runner for the C pretty printer, this file calls c.js
test.c	Sample valid C code examples for testing purposes
Folder lib/ohm/ecmascript	
es5.js	The translator for an implementation of an ECMAScript 5 pretty printer
es5.ohm	The Ohm grammar for a pure ES5 parser
es6.js	Translator extending the ES5 pretty printer to add ES6 features
es6.ohm	Ohm grammar that extends the ES5 grammar to add some ES6 features
pretty.js	Runner for the ECMAScript pretty printer
test.es6	Sample ES6 code examples for testing purposes
test.js	Sample ES5 code examples for testing purposes
Folder lib/ohm/jamscript	
activities.js	Helper file containing code generators for activity calls
activityMatrix.json	Contains a JSON matrix of which activity types are allowed to call other activities.
callGraph.js	Call graph object and utility functions for collecting calls, checking if they are valid, and generating the visualization for the call graph of the program.
jam.js	Starting file for the JAMScript compiler; calls the C and JavaScript JAM translators. Returns the complete output as an object containing both the C and JavaScript code.

Table 5.2: Description of JAMScript compiler files

File	Description
jamc.ohm	Contains the JAMScript extensions of the C parser. Adds activities to the original C syntax. This file extends the grammar file from c.ohm.
jamCTranslator.js	Translator for the JAMScript C parse tree. This file is divided into two sections. The first portion is translators for the new additions to the existing C language, such as activities and jconditional specifiers. The second portion is modifications to the original C translator, such as checking if a variable being assigned to is a JData variable and generating the appropriate code. This file extends the translator from c.js.
jamjs.ohm	Contains JAMScript extensions of the ES5 parser. Adds activity, JData and jconditional declarations to the original syntax. This grammar extends from the ES6 grammar file in es6.ohm, which extends from the grammar in es5.ohm.
jamJSTranslator.js	Translator for the JAMScript JavaScript portion. Similar to the jamCTranslator file, this file sets the actions in the two object variables. jamJSTranslator contains the extensions to the JavaScript translator and es5Translator contains modifications to existing JavaScript translator rules. This file uses the translator defined in es6.js which extends from the translator in es5.js
jCondTranslator.js	Translator for the jconditional specifiers used in activity declaration. This translator generates the code for both the C and JavaScript activities and is inherited in both jamCTranslator and jamJSTranslator.
jdata.js	Used for generating the calls for declaring and assigning to JData variables.

Table 5.2: Description of JAMScript compiler files

File	Description
symbolTable.js	Symbol table object and utility functions. Used for setting and checking identifier declarations. Only JData variables, functions and activities are saved in the symbol table.
types.js	Used for checking if the variable types used in activity and JData declarations are JAMScript compatible and retrieve the requested codes from types.json. This file provides a get method for each possible type of output code request.
types.json	A JSON object that lists the types that are JAMScript compatible. Each type is specified with output codes that are used depending on the situation when generating code.

The following is additional information about files inside of the **lib/ohm/jamscript** folder.

activityMatrix.json – This is a JSON object that lists which activity types are allowed to call other activities types. The file is structured with the following order; for each source language there is a source activity type, then the destination call language and then the destination activity type. For example, to see if an asynchronous C activity can call an synchronous JavaScript activity we check the boolean variable at c.async.js.sync. If set to true the call is allowed and if set to false the call would be rejected at compile time. We perform checks on all calls between activities at compile time and if the user is using an invalid call then the compilation will fail at that point with an error message to the user.

symbolTable.js – Contains the symbol table object that is used to keep track of JData variables, functions and activities used in a JAMScript program. The symbol table supports entering and leaving scopes so that symbols using the same name in different scopes will not interfere with each other.

types.json – A JSON object containing C variable types that are JAMScript compatible and information that is used when generating code for each type. Table 5.3 describes the properties that must be defined to allow using a C variable type as a JAMScript compatible type, as well as an example of the property values to use `int` as a JAMScript compatible type.

Table 5.3: Properties used in types.json

Property	Description	`int` Example
c_pattern	C printf string format pattern	`"%i"`
jamlib	Code used in C jamlib for specifying the variable type	`"ival"`
js_type	JavaScript approximate of the type	`"Number"`
c_code	C shorthand of the type for activity arguments	`"i"`
js_code	JavaScript shorthand of the approximate type	`"n"`
caster	C function for casting non-string types to string	`"get_bcast_int"`
jbroadcast	enum variable for the C type	`"JBROADCAST_INT"`

5.7 Adding Features to JAMScript

In this section we explain how to add a new feature to the JAMScript language with the goal of better understanding how the different portions of the compiler work. For our example we will add jconditionals to JAMScript.

Grammar

To begin adding a feature to JAMScript we must first come up with the syntax. The following is an example of the type of functionality we would like the syntax to be able to allow:

```
jcond {
  ruleA: x > 3;
  ruleB: y < z && node.type == "fog";
}
```

We represent that syntax with the following Ohm grammar:

```
Jconditional = "jcond" identifier? "{" Jcond_entry* "}"

Jcond_entry = identifier ":" NonemptyListOf<Jcond_rule, jcond_logical> ";"

Jcond_rule  = MemberExpr jcond_op MemberExpr

MemberExpr  = MemberExpr "." identifier      -- propRefExp
            | identifier
            | literal

jcond_op    = "==" | ">=" | ">" | "<=" | "<" | "!="

jcond_logical = "&&" | "||"
```

Jconditional will be the starting point for the jcondition declaration. It is defined by beginning with the string **jcond**. This is followed by an optional identifier string (inherited from the base JavaScript definitions) that will be used for the namespace of the jconditional. Between two curly brackets we have 0 or more **Jcond_entry**. Because the rule Jconditional starts with a capital, the number of spaces between its child rules does not matter.

Jcond_entry represents a named jcondition rule. It begins with an identifier string, then a colon, then a **NonemptyListof** (a feature built into the Ohm language). The list takes in at least one **Jcond_rule**, and allows us to chain together multiple Jcond_rules if they are separated by a **jcond_logical**.

Jcond_rule is the boolean condition contained in the jconditional entry. This is a binary operation with a **MemberExpr** on the left- and right-hand sides with a **jcond_op** as the operator in the middle.

MemberExpr specifies what is considered valid expressions in a jconditional. This rule has three alterations and since one of them has a different arity (number of arguments), we must name the alteration with an arity greater than one. We name this rule propRefExp and the rules with only one arity can be left unnamed.

jcond_op is the list of permitted binary comparison operators that we allow in a jconditional expression.

jcond_logical is the list of boolean logical operators we allow inside of jcondition rules.

We now need to hook up the jconditional grammar to our existing grammar. Since jconditional definitions should be inside of our JavaScript files, we add our new grammar to the `jamjs.ohm` file inside of `lib/ohm/jamscript`. Declaring jconditionals should be done at the top-level of the JavaScript file, so we use the previously overloaded JavaScript top-level rule **Declaration** to add support for JAMScript activities:

```
Declaration     += Jconditional
                | Activity_def
```

Translator

With the grammar written we can now write the translator. Inside of `jamJSTranslator.js` we have a `jamJSTranslator` object, each rule in the grammar must have a method with a matching name and the same number of arguments. We add the following functions to be able to parse the tree and generate code:

```
Jcond_rule: function(left, op, right) {
    var code = 0;
    if(left.sourceString === "sys.type") {
      if(op.sourceString === "==") {
        if(right.sourceString === '"dev"') {
          code = 1;
        } else if(right.sourceString === '"fog"') {
          code = 2;
        } else if(right.sourceString === '"cloud"') {
          code = 4;
        }
      } else if(op.sourceString === "!=") {
        if(right.sourceString === '"dev"') {
          code = 6;
        } else if(right.sourceString === '"fog"') {
          code = 5;
        } else if(right.sourceString === '"cloud"') {
          code = 3;
        }
      } else {
        throw "Operator " + op.sourceString + " not
        compatible with sys.type";
      }
    } else if(left.sourceString === "sys.sync") {
      if(op.sourceString === ">=" || op.sourceString
      === "==") {
        if(right.child(0).ctorName === "literal" &&
        Number(right.sourceString) > 0) {
            code = code | 8;
        }
      }
    } else if(left.child(0).ctorName !== "literal" ||
    right.child(0).ctorName !== "literal") {
        code = code | 16;
      }
      return {
        string: "jcondContext('" + left.sourceString +
        "')" + op.sourceString + ' ' +
        right.sourceString,
        code: code
      };
    },
  Jcond_entry: function(id, _1, rules, _2) {
    var first = rules.child(0).jamJSTranslator;
    var seperators = rules.child(1);
    var rest = rules.child(2);
    var code = first.code;
    var string = first.string;
    for (var i = 0; i < rest.numChildren; i++) {
      string += ' ' +
      seperators.child(i).sourceString + ' ' +
      rest.child(i).jamJSTranslator.string;
      code = code |
      rest.child(i).jamJSTranslator.code;
    }
    return {
      name: id.sourceString,
      string: string,
      code: code
    };
  },
```

```
Jconditional: function(_1, id, _2, entries, _3) {
  var output = "";
  var namespace = "";
  if(id.numChildren > 0) {
    namespace = id.sourceString + ".";
  }
  for(var i = 0; i < entries.numChildren; i++) {
    var entry = entries.child(i).jamJSTranslator;
    output += "jnode.jcond.set('" + namespace +
    entry.name + "', "         + "{ source: '" +
    entry.string + "', code: " + entry.code + "
    });\n";
    jCondTranslator.set(namespace + entry.name, {
      source: entry.string,
      code: entry.code });
  }
  return output;
}
```

The methods must have the same number of arguments as the matching rules in the grammar. Ohm generates a CST when parsing, therefore, any terminal characters that we use in rules, such as `"("` or `")"`, must also be an argument in the function. A technique for notating that these variables contain no useful information for code generation is to name them beginning with an underscore.

Jcond_rule has three arguments, all of which we will use. **left** and **right** are both sides of a boolean equation, while **op** is the operator for the equation. We do special processing to determine if the string `sys.type` appears in the equation, as we will use this information to generate a bitmap code that is used internally by JAMScript when performing calls. We then return an object containing the generate source code and the bitmap code.

Jcond_entry has two arguments that we will use during code generation, **id** and a NonemptyListOf of JCond_rule using jcond_joiner as separators between each Jcond_rule. NonemptyListOf is a built-in rule in Ohm that contains three children; the first child is the first element in the NonemptyListOf, the second child is a list of the separators used between each node in the NonemptyListOf, and the third child is the list of remaining children in the NonemptyListOf. We build our output by parsing the first child node and then using a loop we go through all remaining children and separators in the list. We use the `.jamJSTranslator` property on each Jcond_rule. This will continue walking down the parse tree while performing the action specified in the Jcond_rule function inside of the jamJSTranslator object. The Jcond_rule function returns an object, so when we use jamJSTranslator we get an object containing two properties: string and code.

Jconditional has two arguments that we will use for code generation: **id**, which is an optional identifier, and **entries**, which is zero or more Jcond_entry. To check if id is specified we check if there is a child; if there is then the id is specified and the child contains the identifier. As we only need the string of the id, we can use the `sourceString` parameter and skip traversing into the child node. We then loop over the Jcond_entry children, parsing each child to generate output code and setting values in a jCondition hash map. Parsing a Jcond_entry returns an object and we can access the information returned by calling its properties. The Jconditional method returns a string containing the generated code.

We do not need to define translator rules for MemberExpr, jcond_op or jcond_logical because we are only accessing the string values stored in them. When we are using these nodes, we can get the necessary information from them by using the sourceString property instead of recursively calling the `jamJSTranslator` and having to define an action for each node.

5.8 Sample Compilation

In this section we go step by step through the process that the jamc compiler takes when processing a JAMScript program. Our sample program will alternate activity calls between the C side and the JavaScript side, incrementing a counter that counts the total number of calls. The program also prints the value of the counter when it returns to the C side and save the value to a logger.

JavaScript file:

```
#include <stdio.h>

void ping(int);

jasync function pong(int count) {
  x = count;
  printf("%i\n", count);
  ping(count+1);
  return;
}

int main() {
  ping();
  return 0;
}
```

C file:

```
#include <stdio.h>

void ping(int);

jasync function pong(int count) {
  x = count;
  printf("%i\n", count);
  ping(count+1);
  return;
}

int main() {
  ping();
  return 0;
}
```

The compiler starts by reading in both the C file and JavaScript files, checking to make sure both files are specified. The compiler then sends the C input file for preprocessing. The include statement of `stdio.h` will be saved in an array and a marker is placed into the code after the include statements and before the user's code. The Clang preprocessor is then ran on the C input file using fake headers as a replacement for the standard libraries. This means that no code from the actual `stdio.h` will be generated into the preprocessed source file. If there were other libraries used then they would be preprocessed normally and the appropriate code would be placed into the generated source file.

The JAMScript compiler then calls the Ohm compiler with preprocessed C file and the original JavaScript input file. The Ohm compiler starts by parsing the JavaScript file by matching rules and creating a parse tree. The parse tree is then sent into to tree walker, which walks through the parse tree to generate output code. A symbol table hash map is used to keep track of JData variables, activities and functions defined by the user. The tree walker works by checking the name of the rule that was added to the parse tree and looking for a matching action with the same name. As we walk through the parse tree, the JData declaration of variable `x` will be stored in the symbol table with its type information. The activity declaration of `ping` will also be stored, but no code for the activity is generated until we process the C file and see if there is a matching prototype. Delaying code generation allows us to generate code with the appropriate C data types for the parameters and return type. Whenever we see a function call we store it in the call graph object. When we finish walking through the JavaScript tree, we then move onto the C side. The symbol table is reset so only the activities and JData variables remain and is then passed to the C walker and call graph. In our example, the symbol table at this point will only contain the activity `ping` and the JData variable `x`.

The preprocessed C input file is then sent to the parser, and the generated parse tree is walked through to create the output source code. The first statements parsed by the C translator will be the code generated by the preprocessor; when walking through these statements we look for the marker that was placed into the code during the preprocessing step. This marker tells us to reset the code generation output buffer, as we know all code above the marker is code created by the preprocessor and will be generated again later. During translation we find a matching prototype for the JavaScript activity `ping`, so we generate the C code for the remote activity call and the JavaScript code for the activity. When we walk through the code we check simple assignments statements, such as `as x = count`, to see if the variable on the left-hand side matches a previously declared JData variable. Since `x` is a JData variable, we check if it is assignable for its jdata type and source language. In this case `x` is a JData variable and a logger on the C side, so we replace the statement with the code necessary to write to Redis. The function `main` is renamed as the function `user_main` and a new main method is generated the will perform some automatic tasks before calling the user_main function to begin the user's JAMScript program.

When the C compiler finishes, we combine the JavaScript output from the JavaScript compiler and the JavaScript activity code generated by the C compiler into one JavaScript file. We then take the C code generated by the C compiler and add back the original include statements, in this case `stdio.h`. We then use the Clang to compile the C code. The final jxe executable is created by zipping the compiled C executable and the generated JavaScript code file.

Chapter 6

Potential Programming Patterns

Using the features that JAMScript provides, we can easily implement many new and interesting design patterns. In this section we discuss a number of potential programming patterns and show how they can be implemented in JAMScript.

JAMScript is a language designed to program a cloud, fog and device resource stack. It is designed towards data-intensive applications that involve IoT in large-scale settings, such as smart cities. In this section, we discuss the different programming patterns that are enabled by the JAMScript design. The programming patterns show how the built-in language constructs can be used to achieve the example objectives. In a cloud, fog and device resource stack, one of the problems to address is function placement. For instance, by placing function execution closer to the device we could provide better performance or enable offline processing. One downside of such function placement is the lack of shared information, in applications such as machine learning sharing training operations would reduce duplicated computations.

Another concern with data-intensive IoT applications is controlling data flows so that networks are not inundated by high-rate data updates from the devices. Placing data transformation functions closer to the data sources can be used to reduce the impact of high-rate data updates. The JAMScript design allows data reduction transformations to be applied on the data streams at the device while more processing takes place in the fog or cloud. The ability to decouple the data reduction transformations from the bulk of the computations enables the data filtering pattern in JAMScript.

While fogs bring edge-processing to IoT, they also introduce several problems. One problem is fault tolerance; which determines how devices should reorganize when fogs fail. In a cloud, fog and device resource stack, failures could occur at any of the three levels. In this chapter, we consider fog failures only. We assume cloud failures are masked by existing cloud-specific techniques, such as active state-machine replication approaches. We focus on the ability to reorganize the device-to-fog associations once a fog failure is detected. We present a fog fail-over pattern that can be employed by the programs to tolerate fog failures.

Another problem that is interesting in a cloud, fog and device resource stack is scaling or modifying device to fog associations as their relative numbers change. Unlike clouds, fogs are not elastic because they are small-sized resource pools that don't have much spare capacity. Therefore, device-to-fog associations need to adapt according to the location of the devices, such as with mobile devices, and the available number of fogs. The most important factor to consider when designing for this problem is that devices are served by the fogs in the best possible manner by taking the fog's load and location into consideration.

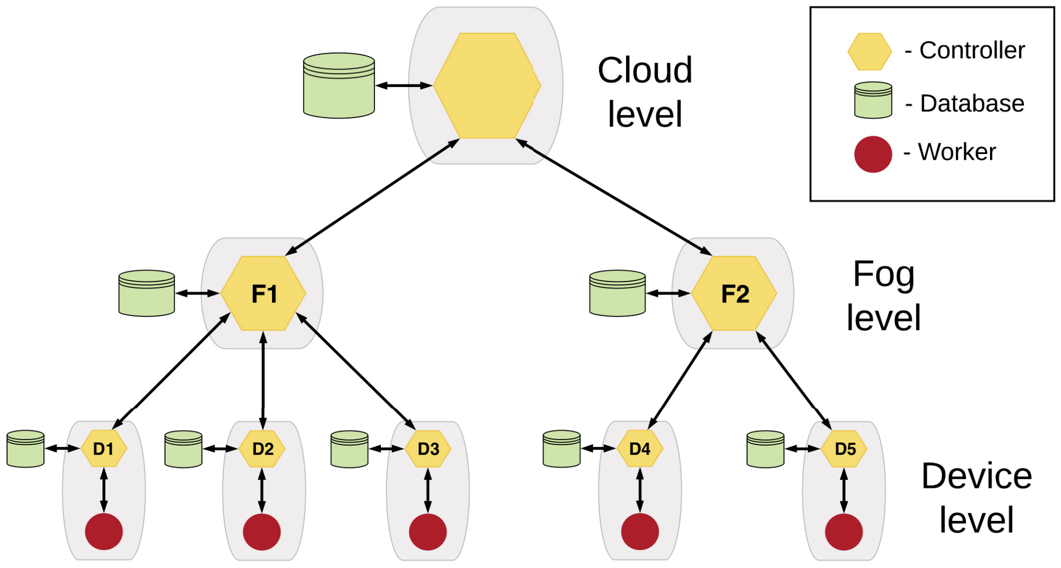

Fig. 6.1 JAMScript Sample Deployment Layout

6.1 Patterns for Edge-Oriented Computing

One of the features designed into JAMScript to support edge-oriented data processing is a variation of the pipes-and-filter pattern for the edge. The goal is to support many concurrent pipe flows at different edge servers where the exchanging applications are co-resident. The concurrent pipe flows should increase the overall throughput of data exchanged between the applications, as well as minimize the disruptions in the data exchanges between applications due to device-fog or fog-cloud disconnections.

Fig. 6.1 shows an example of a JAMScript deployment layout. The controllers represent the J nodes, workers represent the C nodes and databases are the JData Redis deployments. A cloud can have multiple fogs connected, and each fog can have multiple devices connected. For clarity, this diagram shows each device with one J node and one C node but it is possible for a device to have multiple C nodes.

6.2 Function Placement Pattern

When a controller calls a sub-controller, we could execute the call at any of the possible sub levels. For example, a call from the cloud could be executed at the fog or device. Similarly, when a worker calls a controller, we could execute that call in the device, fog, or cloud. The function placement pattern determines the location level where the call actually executes.

The function placement can be either static or dynamic. With static placement, we know at compile time where the function is going to run. We have either explicitly specified the location to run the function on the device, fog or cloud or we have left it as the default policy. The other option is to determine the location using dynamic rules, which is a data dependent approach for determining the execution location for a function.

Static

In the static placement pattern, we use the conditional execution structure, jcondition, to specify at which level functions should run. Knowing where a function will run when writing the program allows the ability to fine-tune the way the data will be handled, such as accounting for the different loads and transmission times that will be encountered. In applications where each J node must be able to respond quickly, it would be beneficial to force execution at the device and not allow for the function to be executed in the fog or cloud. For example, in an autonomous vehicle situation where the J node is performing decisions about what actions the vehicle should make based on the input from C nodes connected to sensors. In this case, if the J node fails on the device or is overloaded and not able to keep up in real time anymore it is best for the application to stop the vehicle instead of trying to rely on the fog or cloud for instructions, as these decisions need to be made with the smallest delays possible and should not be at risk of network problems.

```
jcond {
  deviceOnly: sys.type == "device";
}

jasync function {deviceOnly} deviceOnlyActivity {
  // Code to execute
}
```

In this example we use a jcondition rule to ensure that the activity will only run on the device. The rule **deviceOnly** is defined in the jcond block and then used when defining the activity. Because activities will always try to execute at the lowest level in the hierarchy as possible, it is not necessary to explicitly state that an activity should run on the device, the device will be the default level to run at if no level is specified. This means that having no jconditions at all would fit the pattern of static function placement, as it can be determined at compile time what level on the hierarchy each function will run.

Dynamic

We can apply a dynamic function placement pattern by using the logger and broadcaster to express conditions based on which the function execution location will be decided. An example where this could be useful is when the user would prefer to have calculations performed with minimal latency but would accept a delayed result if necessary. The edge will be able to provide results quicker when the load is low, but when the load on the edge is high the cloud will be able to dynamically provide the same results. To program this example a logger variable is set at the fog by pushing the current CPU load and a broadcaster is set at the cloud to a load threshold. The function to execute would have a conditional rule to run on the fog if the current load, as reported by the logger, is below a certain value. If the current load exceeds that value then the request will go to the cloud for execution.

```
jcond {
  loadCheck: sys.type == "fog" && load < 50;
  cloudRun: sys.type == "cloud";
  fogRun: sys.type == "fog";
}

jdata {
  int load as logger;
}

jasync function {loadCheck || cloudRun} loadBalancedActivity {
  // Code to execute
}

jasync function {fogRun} setFogLoad(curLoad) {
  load = curLoad;
}
```

In this code example we use two jcondition rules. The first rule **loadCheck** will only run on a fog, and uses the logger variable **load** to check if the current load is below a threshold. The load variable can be periodically updated by calling the **setFogLoad** function, which uses the condition **fogRun** to limit execution to the fog. Using the jcondition rule **cloudRun** allows the loadCheck activity to run on the cloud. We use a logical or to join the two rules when defining the function, ensuring the function will always run. Activities attempt to run on the lowest level of the hierarchy as possible; if the loadCheck condition passes the activity will run on the fog, and if it fails the activity will run on the cloud.

6.3 Data Filtering Pattern

Data filtering is a pattern for reducing network bandwidth usage by transforming data at the device or fog. JAMScript provides the logger construct to push data into the network, this can then be filtered by rewriting the data streams at the device or fog. Data filtering is performed by application specific functions provided by the programmer.

We can utilize the data filtering pattern to create a database of the temperature values for a city based on the temperature sensors attached to connected devices. In this situation, it would be redundant to store the information from every temperature in the city. Data filtering could be utilized to reduce all the separate readings to a single value. For example, if we equipped vehicles with multiple temperature sensors, we could have each sensor as an independent C node writing to a logger. An on-device J node could compare the results of all sensors and perform a decision about which temperature reading is most likely to be correct. In our example we will pick the one with the lowest temperature, as this sensor is likely to be the least affected by sunlight. We can then send this data to the fog, which would perform a second data filtering. As each fog is responsible for a limited physical space, it would be redundant to store the temperature readings from each device connected. An average of all reported temperatures from the devices can be taken, to account for different calibrations in the sensors, this can then be sent to the cloud for logging.

```
jdata {
  int tempSensors as logger(device);
  int vehicleTemps as logger(fog);
  int areaAverage as logger(cloud);
}

jcond {
  deviceOnly: sys.type == "device";
  fogOnly: sys.type == "fog";
}

jasync function {deviceOnly} vehicleTempFinder() {
  var connectedDevices = tempSensors.size();
  var lowestValue = tempSensors[0].getLastValue();
  for(var i = 0; i < connectedDevices; i++) {
    var sensorTemp = connectedDevices[i].getLastValue();
    if(sensorTemp < lowestValue) {
      lowestValue = sensorTemp;
    }
  }
  vehicleTemps = lowestValue;
}

jasync function {fogOnly} areaAverageCalculate() {
  var connectedVehicles = vehicleTemps.size();
  var sum = 0;
  for(var i = 0; i < connectedVehicles; i++) {
    sum += vehicleTemps[i].getLastValue();
  }
  areaAverage = sum / connectedVehicles;
}
```

Here we show the JavaScript code for the example described above. On the C side, not shown, we would have each node writing to the device logger **tempSensors** with the temperature reading of a sensor in a function. The **vehicleTempFinder** activity will run on the device J node because of the **deviceOnly** jcondition rule. This activity goes through the last logged state of the tempSensors logger and finds the sensor with the lowest value. It logs the result to a fog logger **vehicleTemps**, which stores the lowest value for all the vehicles in the fog. A fog only activity, **areaAverageCalculate**, takes the average of all reported vehicle temperatures for that fog and saves it to the cloud logger **areaAverage**.

Another form of data filtering is to use the technique of selective logging. Instead of all data from a node being sent to a parent node to decide what needs to be saved, we could select which data is important before it is sent and only send that. In this case we decide at a parent level, such as the cloud, what criteria we are looking for in a device and tell the device to turn on additional logging. For example, we can create a database of how weather anomalies can affect engine performance using selective logging. The vehicular devices would log their current position using GPS coordinates and the cloud would be connected to a live database of weather anomalies, such as smog. The cloud could then tell any vehicle that is within the GPS coordinates of a weather anomaly to turn on complete engine sensor logging.

```
jdata {
  int engineLog as logger(cloud);
  struct curPos {
    int lat;
    int lng;
  } as logger(device);
  struct anomaly {
    int latMin;
    int latMax;
    int lngMin;
    int lngMax;
  } as broadcaster;
}

jcond {
  deviceOnly: sys.type == "device";
  anomalyZone:
    curPos.lat > anomaly.latMin && curPos.lat < anomaly.latMax
    && curPos.lng > anomaly.lngMin && curPos.lng < anomaly.lngMax;
  cloudOnly: sys.type == "cloud";
}

var currentEngineTemp;

jasync function {deviceOnly && anomalyZone} startLogging() {
  setInterval(function() {
    engineLog = currentEngineTemp;
  }, 500);
}

function broadcastAndLog(newAnomaly) {
  anomaly = newAnomaly;
  startLogging();
}
```

In the code example above, we use a logger struct to record the current position of the vehicles and a broadcaster struct to record the bounding box of an anomaly we want to track. When an anomaly is detected the function **broadcastAndLog** is called with an Object containing the bounding box of the anomaly. The function will broadcast the Object and then call the **startLogging** activity. The activity will only run on device, representing the vehicle, and check if the vehicle's current position, as recorded into the **curPos** logger, is within the anomaly zone by using jcondition comparison. If the location is within the area, the vehicle will start saving the current engine temperature into the **engineLog** logger every 500 milliseconds.

6.4 Fog Fail-Over Pattern

The Fail-Over at the Edge pattern deals with designing applications to handle failures at the edge to minimize the distribution in the application. When a fog node fails the runtime will automatically try to connect to a new fog, but if the failed fog was performing an action for the device then all progress will be lost. In Fig. 6.1, if the fog F1 fails then all the devices that were previously connected would try to connect to the nearest fog in range (F2).

We can use the fog fail-over pattern in situations where the device is using the fog to perform long calculations, in these cases if the fog fails we would like to avoid restarting the calculations from the beginning. We could handle this by periodically saving the progress of the calculations in a logger at the fog. If the fog would then crash, the device would automatically reconnect to a new fog but all progress would be lost. We could use the cloud to detect that progress has restarted, based on the data being saved to the logger, and could push out the last saved update from the failed fog to the newly connected fog using the broadcaster.

```
jdata {
  int saveState as logger;
  int recoveryState as broadcaster;
}

jcond {
  fogRun: sys.type == "fog";
  cloudRun: sys.type == "cloud";
}

var computation;
jasync function {fogRun} fogCompute() {
  // computation is assigned to with a progress update

  saveState.log(computation);

  var oldProgress = recoveryState.getLastValue();
  if(oldProgress > computation) {
    computation = oldProgress;
  }
}

var saveStateLog;
jasync function {cloudRun} cloudCheck() {
  var newUpdate = saveState[0].getLastValue();
  if(newUpdate) < saveStateLog) {
    recoveryState.broadcast(saveStateLog);
  } else {
    saveStateLog = newUpdate;
  }
}
```

In this code sample we show a simple version of a fog-fail over pattern example. We assume there is only one device, one fog and one cloud. We use an int as the jdata variable type to save progress for simplicity in this example, JData would likely need to be a structure variable to handle more complicated data storage. The **fogCompute** activity performs a calculation that is not shown, and saves the result to the cloud logger **saveState**. The cloud runs an activity, **cloudCheck**, that checks if the state of the logger is further along than the previous result. If the result is less than the previous result then the cloud can assume that the fog has failed and a new fog was brought in, restarting the computation. The cloud then broadcasts the last recorded value back to the fog. The fog checks each time it writes to the log if the cloud has broadcasted a new greater value, showing that there was a previous fog further along in computation, and replaces its progress with the broadcasted value. When we add more devices and fogs to this example, we would have to include an identifier code in the logger so that the cloud would know which fog it is comparing progress with.

6.5 Edge Covering Pattern

The Edge Covering Pattern represents the method to handle device nodes transitioning between connected fogs. In Fig. 6.1 this would be a device, such as D3, determining that it would be more appropriate for it to be connected to fog F2.

For example, the devices could represent moving objects, such as people carrying smart devices, and the fogs could each represent a room running a fog machine. When the user carrying a device changes rooms, they might still have a connection to the previous room even though the current room has a better connection. We could trigger the system to rearrange in a more optimal layout by periodically disconnecting devices from their fogs. This would cause the devices to automatically try reconnecting to the best available fog, and creating a new more optimal layout.

6.6 Pipes and Filters at the Edge Pattern

Using the piping and filtering provided by JData flows, we can have multiple applications deployed on a single fog node communicating with each other. This allows the applications to pass data back and forth, creating a pattern that we can utilize. We can separate functionality into different applications that we can selectively run on different devices, depending on the required functionality.

For example, we could have an application that runs on smart lighting devices in a room and a separate application that runs on smartphones. When the user enters the room, the phone will connect to the same fog that the lighting is connected to, allowing the two applications to communicate with each other at the fog level using flows. When the user requests a change of lighting on their phone, the message would be sent to the fog using a logger. The fog would then use a flow to send a message to the application controlling the state of the lights, which would use a broadcaster to send a message to all lights in the room to change their state.

Chapter 7

Experimental Results

In this chapter we analyze the runtime performance of various JAMScript components by focusing on the time delays that occur when using these features. The time measurements were calculated using the monotonic clock on C, and the JavaScript library posix-clock[1] to access the monotonic clock through a C++ library.

Experimental Setup: All testing was performed on a single computer, simulating a JavaScript client device and C device nodes. For the purposes of these tests, a single machine setup was chosen as a way to analyze the peak performance of JAMScript. We have previously performed experiments with JAMScript in Docker containers and with Raspberry Pis for multi-node experiments, but in this section we are would like to minimize all possible delays that are not inherent in the JAMScript implementation, such as networking delays. The machine used for testing had a 2.7 GHz Intel Core i5 dual-core processor and 8 GB of DDR3 RAM.

7.1 Activity Calls

In this section we measure the time delay of executing JAMScript activities, this is the extra time cost of launching a remote activity compared to running a function locally. All activity calls in the section are remote function calls between languages. Calls specified as originating from the C side are calling activities on the JavaScript side, and calls specified from the JavaScript side are calling activities located on the C side.

[1]https://github.com/avz/node-posix-clock

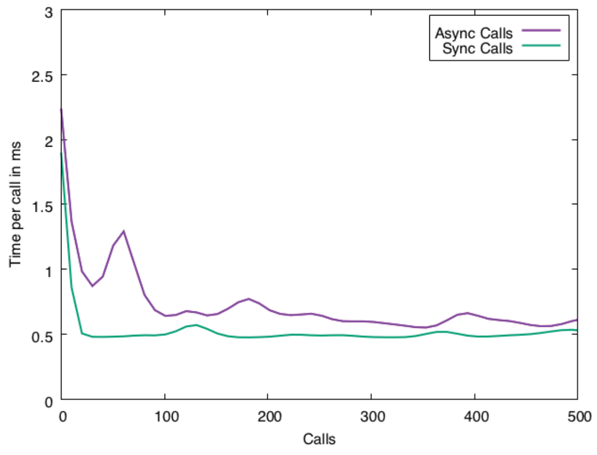

Fig. 7.1 Initial Call Time

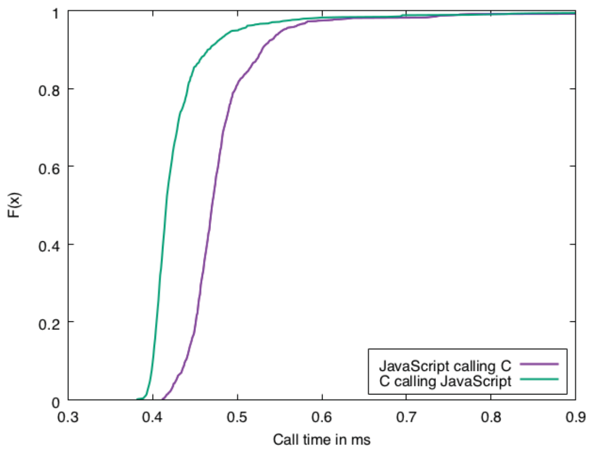

Fig. 7.2 C and JavaScript Call Time

Our initial testing found that JAMScript activities have a longer delay before executing when the program has recently launched and the total number of activities called is low. This is likely caused by the Just-in-Time compiler that Node.js V8 engine is using; when code is repeatedly executed it becomes more thoroughly optimized by the compiler. To account for this in later performance tests we first run 1,000 asynchronous activities and 1,000 synchronous activities before continuing the testing. In Fig. 7.1 we measure the time between calling an activity and when the code in the activity begins executing for the first 500 calls of each activity type. Our results show that the time to call an asynchronous activity will become approximately 5 times faster than at the first execution, taking around 300 calls to reach optimal performance. When we run the synchronous activities after the asynchronous activity calls complete, it only takes a few calls to reach optimal performance. This shows that most of the JAMScript library code is shared between the synchronous and asynchronous activities and that the optimizations made by the compiler are shared by the different types of activities in JAMScript.

In Fig. 7.2 we compare the time to run 1,000 asynchronous activity calls when launched from JavaScript to call C code and compare it to the time to run 1,000 asynchronous activities that are launched from C side to call JavaScript code. The time is measured as the difference between when the activity was called to when the code inside the activity begins executing. Our results show that the JAMScript code is slightly faster when launched from the C side. This is likely due to the fact that there is more processing being performed on the side that launches an activity call and that C code is, in general, faster than JavaScript.

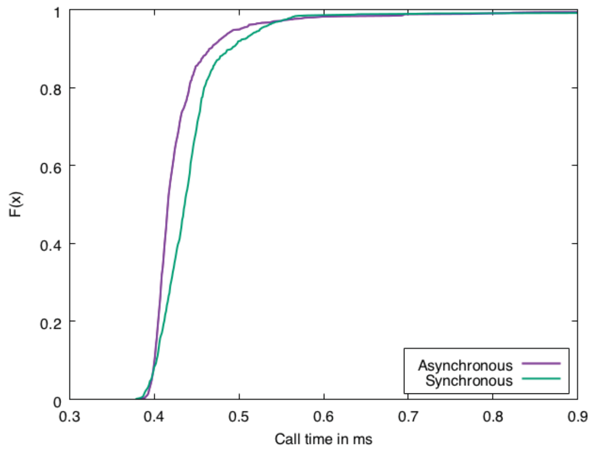

Fig. 7.3 Sync and Async Call Time

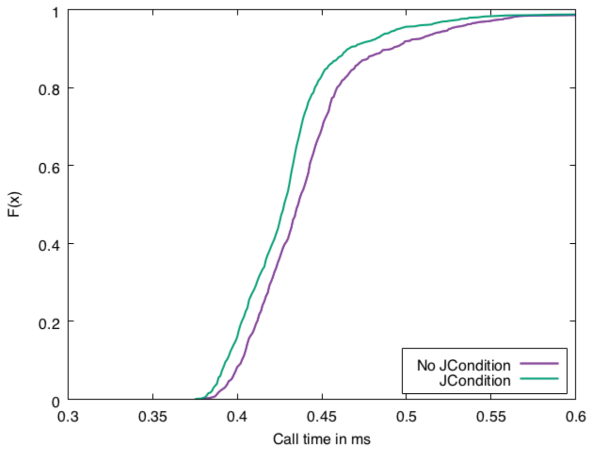

Fig. 7.4 Jcondition Call Time

In Fig. 7.3 we compare the time to run 1,000 asynchronous activity calls to the time to run 1,000 synchronous activity calls, with all calls being launched from the C side. The time is measured as the difference between the activity was called to when the code inside the activity begins executing. We see that the time to launch an asynchronous activity is slightly faster than launching a synchronous activity. This is likely be due to the asynchronous code having less overhead, as the calling code does not need to look for a return value.

In Fig. 7.4 we compare the time to run an asynchronous activity with and without a jcondition rule applied. This allows us analyze the additional overhead that running a jcondition adds to the runtime. The code was launched from the JavaScript side and we recorded the time it takes for the code inside the activity to begin execution. The condition tested was a simple binary equation that will always evaluate to true to minimize the amount of processing that will be done to evaluate the rule. The results show that there is a slight performance increase when using the jcondition, which may be due to increased caching.

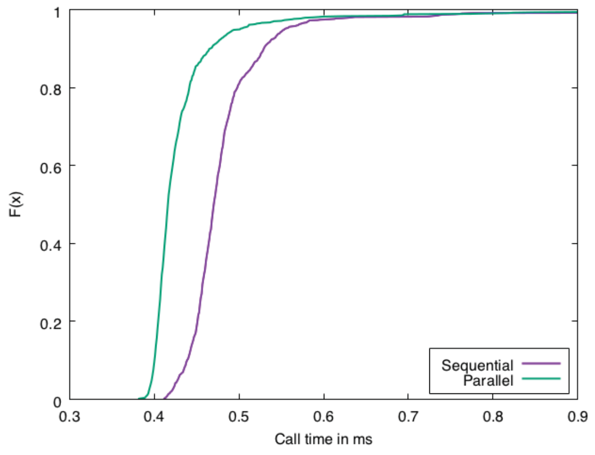

Fig. 7.5 Parallel and Sequential Call Time

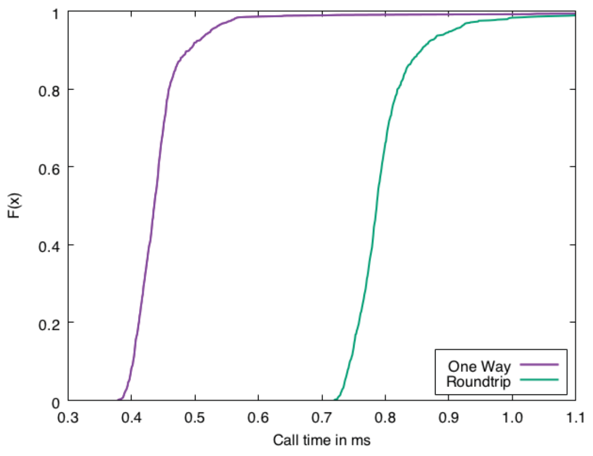

Fig. 7.6 Sync Roundtrip Call Time

In Fig. 7.5 we compare the time to run 1,000 asynchronous activities sequentially compared to running them in parallel when launched from a C client. In the sequential test we launch an activity, wait for activity to complete, and then launch the activity again. In the parallel test we launch all 1,000 activities without waiting for the previous activity to finish. We recorded the time for the code inside the activity to begin execution and our results show that there is a performance advantage to launching activities in parallel.

In Fig. 7.6 we compare the time to start executing code inside of a synchronous activity and the time for an empty synchronous activity to return a value, when called from a C client. We see that returning a value from a synchronous activity is faster than twice the time to launch an activity. This indicates there are performance benefits, due to less overhead, of using synchronous return values instead of using a separate activity to return a value.

7.2 Reading and Writing to JData

In Fig. 7.7 we compare the performance of a JData logger variable as we increase the number of C nodes connected. The test was performed by inserting a new value into a logger variable and recording how long it took for the JavaScript side to register a change in the variable's contents and then repeated 1,000 times. We doubled the number of C clients after each test and then ran the test on each client at the same time. The results show that as we increase the number of C devices there is a corresponding increase in the time it takes for the change in variable contents to occur. This likely because each client is trying to write to a single Redis server at the same time, creating a bottleneck. There is also an approximately 10 percent chance that the update takes substantially longer time than ordinary, which could be because of the way we batch and send updates to the Redis server.

In Fig. 7.8 we compare the performance of the JData broadcaster as we increase the number of C nodes connected to the J node. Our results show that from 1 to 4 C devices the performance of the broadcaster does not change significantly but at 8 C devices there begins to be a performance penalty. This may show that increasing the number of devices does not decrease the performance of the broadcaster directly, but after a certain number of devices are connected there is a resource limitation, possibly caused by simulating too many devices on a single machine.

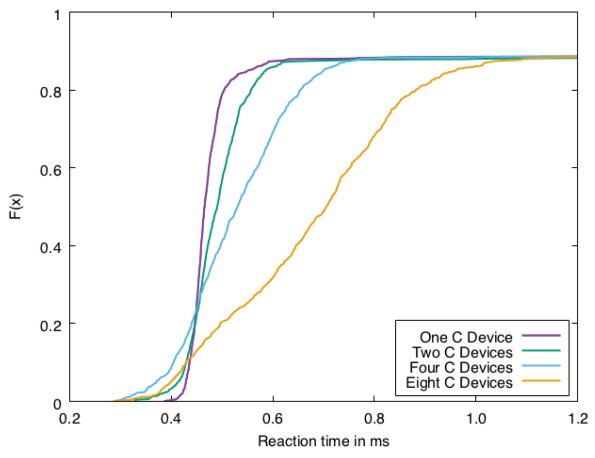

Fig. 7.7 Logger Reaction Time

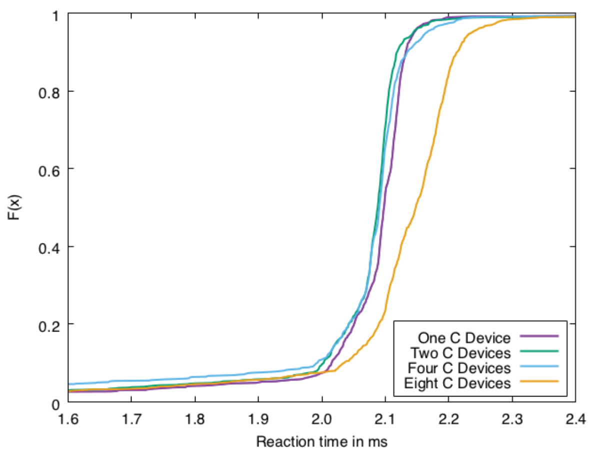

Fig. 7.8 Broadcaster Reaction Time

Chapter 8

Related Work

Software-Defined Programming

Software-defined networking has inspired a host of software-defined programming frameworks for IoT [14, 15, 16], edge [17], fog [18, 19] and cloud computing [20]. A software-defined IoT framework encompassing software-defined storage [21] and software-defined security [22] was developed in [14]. The framework's model includes the following layers: the physical layer consisting of the physical devices in the system, the middleware layer where the IoT and software-defined controllers reside and the application layer. A similar three-layered architecture was proposed in [15] for smart urban sensing.

MIST [19], a software-defined fog system leveraging software-defined mobility [23] for real-time surveillance was developed to provide device mobility and change adaptively based on context. Nodes are classified as being controllers, hosts or switches to enable seamless execution even in the presence of frequent changes in membership. A runtime framework for managing software-defined IoT clouds was proposed in [20]. The goal of the framework is to abstract dynamicity and scalability issues from the management of software-defined IoT cloud systems. The framework provides an automated and fine-grained central control and autonomous IoT cloud resources aimed at improving system efficiency.

Cloud Programming

Orleans [24] is a software framework for building scalable and reliable cloud applications. Orleans uses actor-based distributed components called grains that consist of isolated units.

Grains communicate asynchronously through message passing and use a single-threaded execution model. In cases of high load and lower system throughput, Orleans creates multiple instances of a busy grain to handle multiple simultaneous executions. The consistency model in Orleans is achieved using optimistic and atomic transactions that are isolated. One of the main design goals of Orleans is that the runtime handles system level attributes such as scalability, reliability, fault tolerance and durability while application developers only focus on the application logic.

Dripcast [25] 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. Dripcast implements transparent Java remote procedure calls and a mechanism to read, store and process Java objects in a distributed, scalable data store.

IoT Programming

Mobile Fog [26] is a platform as a service programming framework for IoT. An interface with a single application code is exposed that allows dynamic scaling. A Mobile Fog application consists of processes that cover defined geographical locations. Processes are connected in a three-tier hierarchical architecture and are distributed on computing elements such as cloud, fog and edge devices. Mobile Fog assumes that fog computing infrastructure nodes are placed in the network and a programming interface is provided by the fog. Mobile Fog handles scaling by making application developers specify the scaling policy at each hierarchical level. Load balancing is based on creating on-demand fog instances at the same level as an over-loaded fog instance.

Calvin [27] is a framework 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. These phases are supported by the runtime, APIs and communication protocols. The platform dependent part of the Calvin runtime manages inter-runtime communication, transport layer support, abstraction for I/O and sensing mechanisms. The platform-independent runtime provides an interface for the actors, the scheduler of the Calvin runtime also resides in this layer.

The Calvin runtime supports multi-tenancy, once an application is deployed actors may share the runtime with actors from other applications.

Simurgh [28] provides a high level-programming framework for IoT application development. The framework supports exposing IoT services as RESTful APIs and composing the 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” is the network discovery and registration broker that listens to incoming connection requests from the devices’ and handles them. The “platform layer” stores information about things and services, manages flow design and composition and handles requests. An API mediator assists programmers to expose their applications through RESTful APIs. 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.

Polyglot Programming

GraalVM [29] supports a system for developing high performance applications in Java with the ability to use multiple programming languages in one program. Graal applications can be written in any combination of Java, JavaScript, Python, Ruby, Scala, and all LLVM compatible languages, such as C, C++ and Rust. GraalVM’s support for multiple languages is based on earlier work from TruffleVM [30]. TruffleVM allows access to foreign language functions and objects while keeping the code to use these features to a minimum. For each supported programming language, TruffleVM uses a language implementation for translating the input code into a common intermediate representation. The output is then dynamically compiled by the Graal compiler and executed by GraalVM in the Java HotSpot Runtime. The TruffleVM polyglot design was later incorporated into GraalVM with support for more languages by adding additional Truffle implementations. GraalVM supports all LLVM compatible languages by utilizing Clang to generate LLVM bitcode and then using the bitcode to dynamically compile the program in the HotSpot VM. GraalVM can also compile Java code ahead-of-time to run on the custom high performance VM, SubstrateVM.

Mobile Computing Programming

Odessa [31] proposes a mobile programming model where application developers structure applications as data flow graphs. Odessa is best suited for streaming applications where automatic data offloading and parallelism are desired attributes. An edge or connector in a data flow graph represents data dependencies between the vertices, which represent different stages of processing. Processing stages in an application operate independently with no shared states and only communicate through connectors, thus abstracting the programming details and complexity of different processing stages from one another. Odessa's programming design is based on the Sprout runtime [32] which allows developers to write parallel and distributed applications while hiding the complexity.

Programming Frameworks for Distributed Systems

EventWave [33] is an event-driven programming framework for writing distributed applications. EventWave allows a single logical node to be distributed over a number of physical nodes. The notion of atomic events is introduced, this allows application developers to not have to focus on elasticity but rather on the application behavior and logic. It assumes that an application that runs correctly on a few logical nodes will run correctly when the number of nodes increases or decreases.

The eLinda [34] model extends Linda, a coordination-programming model for writing parallel and distributed applications. The Linda model supports a shared memory store called tuple space for communication between the processes of an application. eLinda provides support for broadcast communication. It proposes the "Programmable Matching Engine" (PME) that allows program developers to specify a custom matcher that can be used internally to retrieve tuples from a shared store. The PME has been found to be advantageous for parsing graphical languages and video-on-demand systems.

Programming Frameworks for Intermittent Systems

DINO [35] was proposed as a programming model for addressing the issues of intermittent (partial or repeated) executions that can result in faulty or wrong outputs and cause consistency problems. Long-running computations are semantically divided into shorter tasks through task boundaries inserted by application developers. DINO guarantees that an applications state at the task boundary is consistent with the finished execution of a preceding task. Checkpointing and failure recovery are used to track the states of the execution of an application. Instructions in DINO are executed as transactions and failures are handled by resuming execution at the task boundary before the failure.

A similar programming model for intermittent programs called Chain [2] was proposed. Chain argues that checkpointing as used in [35] is computationally expensive. Therefore, it uses a control flow that is task-based for failure recovery and a memory model that is channel-based which allows data exchange among tasks with high consistency guarantees. A Chain program is a collection of tasks that form a task graph which defines task controls flows (inflow and outflow).

Chapter 9

Conclusions and Future Work

We believe JAMScript is positioned to be an early innovator in the edge computing space. By building JAMScript on top of a combination of languages that already are popular in the developer community we can give programmers a quicker learning period for adopting the language. The learning period of JAMScript for existing C and JavaScript developers is designed to be as minimal as possible by integrating features into existing language structures and syntax.

JAMScript has the ability to implement new edge computing design patterns with minimal effort by taking advantage of built in features. In IoT situations a complex network of nodes can be deployed easily and reorganized automatically to maximize performance. Smart cities can deploy large scale JAMScript applications with vehicular devices running as devices, reconnecting to nearby fogs as they travel between city infrastructure nodes. Smart devices can run JAMScript applications to control home automation systems in a home or business, connecting to different fogs depending on their location. Machine learning applications can potentially benefit from JAMScript as well. Neural Networks can be created using nodes as neurons, allow a system that can share tasks over multiple devices and handle failure of individual nodes without having to restart computations.

In this thesis, we covered the design and benefits of JAMScript. We detailed how the JAMScript compiler works, including background information on how to write an Ohm grammar and parser. The structure and components of the JAMScript compiler architecture were presented. We explained the syntax of JAMScript and demonstrated how to write a JAMScript program. We detailed the features of JAMScript, demonstrating how to make activity calls and use callbacks. We introduced the shared memory store JData, the conditional execution construct jconditional and the streaming data pipeline of flows. We show how to modify the compiler using Ohm to add new features to JAMScript. We created fog computing programming patterns and showed how they can be achieved with minimal effort using the features that JAMScript provides. Finally, we demonstrated the performance of JAMScript with experimental testing.

In the future, JAMScript has the potential to improve in many ways, from schedulers that use call information and machine learning to optimize during runtime to improved discovery services for automatically finding the best nearby fogs and clouds to connect with. On the compiler side, a full symbol table implementation would bring many advantages for static analysis tools that trace the complete usage of a variable through the program. There would also be benefits for JAMScript to move the compiler away from Ohm to another compiler builder technology. Ohm was a good choice during early development to enable rapidly building the grammar and allowing easy expansion of the language when adding new features. Since then, the limitations of Ohm in terms of compiling speed and being able to provide helpful error messages when failing during compilation have become more important.

Appendix A

JAMScript Grammar in EBNF

A.1 C Extension

⟨*program*⟩	::=	⟨*external_decl*⟩+
⟨*external_decl*⟩	::=	⟨*activity_def*⟩
	|	⟨*prototype*⟩
	|	⟨*function_def*⟩
	|	⟨*variable_decl*⟩
	|	⟨*preprocessor_line*⟩
⟨*type_spec*⟩	::=	‘`jcallback`’
	|	/* Inherited C types */
⟨*activity_def*⟩	::=	⟨*sync_activity*⟩
	|	⟨*async_activity*⟩
⟨*async_activity*⟩	::=	‘`jasync`’ ⟨*jcond_specifier*⟩? ⟨*declarator*⟩ ⟨*compound_stmt*⟩
⟨*sync_activity*⟩	::=	‘`jsync`’ ⟨*decl_specs*⟩ ⟨*jcond_specifier*⟩? ⟨*declarator*⟩ ⟨*compound_stmt*⟩

```
⟨jcond_specifier⟩      ::=  ‘{’ ⟨jcond_expr⟩ ‘}’

⟨jcond_expr⟩           ::=  ‘(’ ⟨jcond_expr⟩ ‘)’
                        |   ‘!’ ⟨jcond_expr⟩
                        |   ⟨jcond_expr⟩ ⟨jcond_expr_op⟩ ⟨Jcond_expr⟩
                        |   ⟨id⟩ ‘.’ ⟨id⟩
                        |   ⟨id⟩

⟨jcond_expr_op⟩        ::=  ‘&&’ | ‘||’

⟨function_def⟩         ::=  /* Inherited from C grammar */

⟨variable_decl⟩        ::=  /* Inherited from C grammar */

⟨preprocessor_line⟩    ::=  /* Inherited from C grammar */

⟨compound_stmt⟩        ::=  /* Inherited from C grammar */

⟨declarator⟩           ::=  /* Inherited from C grammar */

⟨decl_specs⟩           ::=  /* Inherited from C grammar */

⟨compound_stmt⟩        ::=  /* Inherited from C grammar */

⟨id⟩                   ::=  /* Inherited from C grammar */
```

A.2 JavaScript Extension

```
⟨program⟩          ::= ⟨declaration⟩+

⟨declaration⟩      ::= ⟨jconditional⟩
                    |  ⟨jdataDecl⟩
                    |  ⟨activityDef⟩
                    |  ⟨jviewDecl⟩
                    |  ⟨declaration⟩
                    |  ⟨statement⟩

⟨jconditional⟩     ::= 'jcond' ⟨identifier⟩? '{' ⟨jcondEntry⟩* '}'

⟨jcondEntry⟩       ::= ⟨identifier⟩ ':' ⟨jcondRule⟩ ⟨jcondList⟩* ';'

⟨jcondList⟩        ::= ⟨jcondJoiner⟩ ⟨jcondRule⟩

⟨jcondRule⟩        ::= ⟨memberExpr⟩ ⟨jcondOp⟩ ⟨memberExpr⟩ (',' ⟨identifier⟩)?

⟨memberExpr⟩       ::= ⟨identifier⟩ '(' ⟨memberExpr⟩ ')'
                    |  ⟨memberExpr⟩ '.' ⟨identifier⟩
                    |  ⟨identifier⟩
                    |  ⟨literal⟩

⟨jcondJoiner⟩      ::= '&&' | '||'

⟨jcondOp⟩          ::= '==' | '>=' | '>' | '<=' | '<' | '!='

⟨jdataDecl⟩        ::= 'jdata' '{' ⟨jdataSpec⟩* '}'
```

⟨*jdataSpec*⟩	::=	⟨*cType*⟩ ⟨*identifier*⟩ ‘`as`’ ⟨*jdataType*⟩ ‘`(`’ (‘`fog`’|‘`cloud`’) ‘`)`’ ‘`;`’
	|	⟨*cType*⟩ ⟨*identifier*⟩ ‘`as`’ ⟨*jdataType*⟩ ‘`;`’
	|	⟨*flow*⟩ ‘`;`’
⟨*jdataType*⟩	::=	‘`broadcaster`’
	|	‘`logger`’
⟨*cType*⟩	::=	‘`struct`’ ⟨*identifier*⟩ ‘`{`’ ⟨*structEntry*⟩* ‘`}`’
	|	‘`char`’ ‘`*`’
	|	‘`char`’
	|	‘`double`’
	|	‘`int`’
	|	‘`float`’
⟨*structEntry*⟩	::=	⟨*cType*⟩ ⟨*identifier*⟩ ‘`;`’
⟨*flow*⟩	::=	⟨*identifier*⟩ ‘`as`’ ‘`flow`’ ‘`with`’ ⟨*identifier*⟩ ‘`of`’ ⟨*identifier*⟩
	|	⟨*identifier*⟩ ‘`as`’ ‘`outflow`’ ‘`of`’ ⟨*identifier*⟩
	|	⟨*identifier*⟩ ‘`as`’ ‘`inflow`’
⟨*activityDef*⟩	::=	⟨*syncActivity*⟩
	|	⟨*asyncActivity*⟩
⟨*asyncActivity*⟩	::=	‘`jasync`’ ⟨*jcondSpecifier*⟩? ⟨*functionDeclaration*⟩
⟨*syncActivity*⟩	::=	‘`jsync`’ ⟨*jcondSpecifier*⟩? ⟨*functionDeclaration*⟩
⟨*jcondSpecifier*⟩	::=	‘`{`’ ⟨*jcondExpr*⟩ ‘`}`’

```
⟨jcondExpr⟩          ::= ‘(’ ⟨jcondExpr⟩ ‘)’
                       |  ‘!’ ⟨jcondExpr⟩
                       |  ⟨jcondExpr⟩ ⟨jcondExprOp⟩ ⟨jcondExpr⟩
                       |  ⟨identifier⟩ ‘.’ ⟨identifier⟩
                       |  ⟨identifier⟩

⟨jcondExprOp⟩        ::= ‘&&’ | ‘||’

⟨JjiewDecl⟩          ::= ‘jview’ ‘{’ ⟨jviewSpec⟩* ‘}’

⟨jviewSpec⟩          ::= ‘beat’ ‘:’ ⟨identifier⟩ ‘;’
                       |  ⟨jviewPage⟩

⟨jviewPage⟩          ::= ⟨identifier⟩ ‘as’ ‘page’ ‘{’ ⟨pageElem⟩+ ‘}’

⟨pageElem⟩           ::= ⟨displayElem⟩
                       |  ⟨controlElem⟩
                       |  ⟨pageName⟩

⟨displayElem⟩        ::= ⟨identifier⟩ ‘is’ ‘display’ ‘{’ ⟨dispSpec⟩+ ‘}’

⟨controlElem⟩        ::= ⟨identifier⟩ ‘is’ ‘controller’ ‘{’ ⟨ctrlSpec⟩+ ‘}’

⟨pageName⟩           ::= ‘name’ ‘:’ ⟨identifier⟩ ‘;’

⟨dispSpec⟩           ::= ‘type’ ‘:’ ⟨displayType⟩ ‘;’
                       |  ‘title’ ‘:’ ⟨stringLiteral⟩ ‘;’
                       |  ‘options’ ‘:’ ⟨stringLiteral⟩ ‘;’
                       |  ‘source’ ‘:’ ⟨identifier⟩ ‘;’
                       |  ‘refresh’ ‘:’ ⟨decimalIntegerLiteral⟩ ‘;’
```

```
⟨displayType⟩          ::= 'graph' | 'scatter' | 'stackedgraph'

⟨ctrlSpec⟩             ::= 'type' ':' ⟨controlType⟩ ';'
                         |  'title' ':' ⟨stringLiteral⟩ ';'
                         |  'options' ':' ⟨stringLiteral⟩ ';'
                         |  'sink' ':' ⟨identifier⟩ ';'

⟨controlType⟩          ::= 'slider' | 'terminal' | 'button'

⟨declaration⟩          ::= /* Inherited from es6 grammar */

⟨statement⟩            ::= /* Inherited from es6 grammar */

⟨functionDeclaration⟩  ::= /* Inherited from es6 grammar */

⟨identifier⟩           ::= /* Inherited from es6 grammar */

⟨literal⟩              ::= /* Inherited from es6 grammar */

⟨stringLiteral⟩        ::= /* Inherited from es6 grammar */

⟨decimalIntegerLiteral⟩ ::= /* Inherited from es6 grammar */
```

References

[1] M. Zaharia, *An Architecture for Fast and General Data Processing on Large Clusters*. Morgan & Claypool, 2016.

[2] A. Colin and B. Lucia, "Chain: Tasks and Channels for Reliable Intermittent Programs," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2016, pp. 514–530.

[3] A. Warth, P. Dubroy, and T. Garnock-Jones, "Modular Semantic Actions," in *ACM SIGPLAN Notices*, vol. 52, no. 2. ACM, 2016, pp. 108–119.

[4] B. Ford, "Parsing Expression Grammars: A Recognition-Based Syntactic Foundation," in *ACM SIGPLAN Notices*, vol. 39, no. 1. ACM, 2004, pp. 111–122.

[5] T. J. Parr and R. W. Quong, "ANTLR: A Predicated-LL(k) Parser Generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[6] R. P. Corbett, "Static Semantics and Compiler Error Recovery," Ph.D. dissertation, University of California, Berkeley, 1985.

[7] A. Warth, P. Dubroy, and T. Garnock-Jones, "Ohm/JS repository," 2014. [Online]. Available: https://github.com/cdglabs/ohm

[8] P. Rein, R. Hirschfeld, and M. Taeumel, "Gramada: Immediacy in Programming Language Development," in *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, 2016, pp. 165–179.

[9] A. Warth and I. Piumarta, “OMeta: an Object-Oriented Language for Pattern Matching,” in *Proceedings of the 2007 symposium on Dynamic languages.* ACM, 2007, pp. 11–19.

[10] N. Heirbaut and T. Van Der Storm, “Two implementation techniques for Domain Specific Languages compared: OMeta/JS vs. JavaScript,” *Master’s thesis, Universiteit van Amsterdam*, 2009.

[11] S. Sanfilippo, “Redis,” 2009. [Online]. Available: https://redis.io

[12] D. Li, H. Mei, Y. Shen, S. Su, W. Zhang, J. Wang, M. Zu, and W. Chen, “ECharts: A declarative framework for rapid construction of web-based visualization,” *Visual Informatics*, 2018.

[13] M. Franz, C. T. Lopes, G. Huck, Y. Dong, O. Sumer, and G. D. Bader, “Cytoscape.js: A Graph Theory Library for Visualisation and Analysis,” *Bioinformatics*, vol. 32, no. 2, pp. 309–311, 2015.

[14] Y. Jararweh, M. Al-Ayyoub, E. Benkhelifa, M. Vouk, A. Rindos *et al.*, “SDIoT: A Software Defined Based Internet of Things Framework,” *Journal of Ambient Intelligence and Humanized Computing*, vol. 6, no. 4, pp. 453–461, 2015.

[15] J. Liu, Y. Li, M. Chen, W. Dong, and D. Jin, “Software-Defined Internet of Things for Smart Urban Sensing,” *IEEE communications magazine*, vol. 53, no. 9, pp. 55–63, 2015.

[16] M. Tortonesi, J. Michaelis, A. Morelli, N. Suri, and M. A. Baker, “SPF: An SDN-based Middleware Solution to Mitigate the IoT Information Explosion,” in *Computers and Communication (ISCC), 2016 IEEE Symposium on.* IEEE, 2016, pp. 435–442.

[17] Y. Jararweh, A. Doulat, A. Darabseh, M. Alsmirat, M. Al-Ayyoub, and E. Benkhelifa, “SDMEC: Software Defined System for Mobile Edge Computing,” in *Cloud Engineering Workshop (IC2EW), 2016 IEEE International Conference on.* IEEE, 2016, pp. 88–93.

[18] N. B. Truong, G. M. Lee, and Y. Ghamri-Doudane, “Software Defined Networking-based Vehicular Adhoc Network with Fog Computing,” in *Integrated Network Man-*

agement (IM), 2015 IFIP/IEEE International Symposium on. IEEE, 2015, pp. 1202–1207.

[19] H. Gebre-Amlak, S. Lee, A. M. Jabbari, Y. Chen, B.-Y. Choi, C.-T. Huang, and S. Song, “MIST: Mobility-Inspired Software-Defined Fog System,” in *Consumer Electronics (ICCE), 2017 IEEE International Conference on.* IEEE, 2017, pp. 94–99.

[20] S. Nastic, M. Vögler, C. Inzinger, H.-L. Truong, and S. Dustdar, “rtGovOps: A Runtime Framework for Governance in Large-scale Software-defined IoT Cloud Systems,” in *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2015 3rd IEEE International Conference on.* IEEE, 2015, pp. 24–33.

[21] F. Wu and G. Sun, “Software-Defined Storage,” *Report*, 2013.

[22] M. Al-Ayyoub, Y. Jararweh, E. Benkhelifa, M. Vouk, A. Rindos *et al.*, “SDSecurity: A Software Defined Security Experimental Framework,” in *Communication Workshop (ICCW), 2015 IEEE International Conference on.* IEEE, 2015, pp. 1871–1876.

[23] L. M. Contreras, L. Cominardi, H. Qian, and C. J. Bernardos, “Software-Defined Mobility Management: Architecture Proposal and Future Directions,” *Mobile Networks and Applications*, vol. 21, no. 2, pp. 226–236, 2016.

[24] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin, “Orleans: Cloud Computing for Everyone,” in *Proceedings of the 2nd ACM Symposium on Cloud Computing.* ACM, 2011, p. 16.

[25] I. Nakagawa, M. Hiji, and H. Esaki, “Dripcast — Server-less Java Programming Framework for Billions of IoT Devices,” in *Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International.* IEEE, 2014, pp. 186–191.

[26] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe, “Mobile Fog: A Programming Model for Large–Scale Applications on the Internet of Things,” in *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing.* ACM, 2013, pp. 15–20.

[27] P. Persson and O. Angelsmark, “Calvin — Merging Cloud and IoT,” *Procedia Computer Science*, vol. 52, pp. 210–217, 2015.

[28] F. Khodadadi, A. V. Dastjerdi, and R. Buyya, “Simurgh: A Framework for Effective Discovery, Programming, and Integration of Services Exposed in IoT,” in *Recent Advances in Internet of Things (RIoT), 2015 International Conference on.* IEEE, 2015, pp. 1–6.

[29] Oracle, “GraalVM,” 2013. [Online]. Available: https://www.graalvm.org

[30] M. Grimmer, C. Seaton, R. Schatz, T. Würthinger, and H. Mössenböck, “High-Performance Cross-Language Interoperability in a Multi-language Runtime,” in *ACM SIGPLAN Notices*, vol. 51, no. 2. ACM, 2015, pp. 78–90.

[31] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, “Odessa: Enabling Interactive Perception Applications on Mobile Devices,” in *Proceedings of the 9th international conference on Mobile systems, applications, and services.* ACM, 2011, pp. 43–56.

[32] P. S. Pillai, L. B. Mummert, S. W. Schlosser, R. Sukthankar, and C. J. Helfrich, “SLIPstream: Scalable Low-latency Interactive Perception on Streaming Data,” in *Proceedings of the 18th international workshop on Network and operating systems support for digital audio and video.* ACM, 2009, pp. 43–48.

[33] W.-C. Chuang, B. Sang, S. Yoo, R. Gu, M. Kulkarni, and C. Killian, “EventWave: Programming Model and Runtime Support for Tightly-Coupled Elastic Cloud Applications,” in *Proceedings of the 4th annual Symposium on Cloud Computing.* ACM, 2013, p. 21.

[34] G. Wells, “Coordination Languages: Back to the Future with Linda,” in *Proceedings of the Second International Workshop on Coordination and Adaption Techniques for Software Entities (WCAT05)*, 2005, pp. 87–98.

[35] B. Lucia and B. Ransford, “A Simpler, Safer Programming and Execution Model for Intermittent Systems,” *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 575–585, 2015.