2016 IEEE 2nd International Conference on Collaboration and Internet Computing

A Programming Language and System for Heterogeneous Cloud of Things

Robert Wenger, Xiru Zhu, Jayanth Krishnamurthy, and Muthucumaru Maheswaran
School of Computer Science
McGill University
Montreal, Quebec
{*robert.wenger, xiru.zhu, jayanth.krishnamurthy*}*@mail.mcgill.ca, muthucumaru.maheswaran@mcgill.ca*

***Abstract*—Cloud of Things (CoT) is a computing model that combines the widely popular cloud computing with Internet of Things (IoT). One of the major problems with CoT is the latency of accessing distant cloud resources from the devices, where the data is captured. To address this problem, paradigms such as fog computing and Cloudlets have been proposed to interpose another layer of computing between the clouds and devices. Such a three-layered cloud-fog-device computing architecture is touted as the most suitable approach for deploying many next generation ubiquitous computing applications. Programming applications to run on such a platform is quite challenging because disconnections between the different layers are bound to happen in a large-scale CoT system, where the devices can be mobile. This paper presents a programming language and system for a three-layered CoT system. We illustrate how our language and system addresses some of the key challenges in the three-layered CoT. A proof-of-concept prototype compiler and runtime have been implemented and several example applications are developed using it.**

***Keywords*-coordination; cloud of things; fault tolerance; heterogeneous systems; programming language;**

I. Introduction

Cloud of Things (CoT) is a computing model that brings together cloud computing and Internet of Things (IoT). IoTs can be resource constrained elements that do not have reliable long-term storage or high-powered processing capabilities, combining them with cloud computing systems can enable new types of applications.

One of the problems of employing cloud computing is the high and often variable latencies of the connection between the devices and compute resources. Because in CoT the devices can generate data at high rate and might expect data processing to happen within a short time duration, the high device-to-cloud latencies could be an impediment. To address this problem, "edge-based" cloud variations such as fog computing and Cloudlets have been developed. These frameworks interpose a computing layer between the devices and clouds to reduce the latencies of accessing the compute resources from the devices, where the data is produced.

Developing programs for such a three-layered CoT is quite challenging. In particular, disconnections and latency variations have to be taken into consideration while implementing fault-tolerance measures. Programming approaches for CoT are primarily based on either remote procedure call (RPC) or representational state transfer (REST). RPC has its prescribed way of implementing fault tolerance while REST leaves fault tolerance to the application. With REST, to handle faults, the programmer needs to provide custom fault handling code that is specific to the application scenario, which has three major limitations: programmability, reusability, and extensibility.

In this paper, we introduce a programming language (referred to as JAMScript) and system (referred to as JAM) for CoT that takes an in-between approach to fault tolerance. The idea is to build basic fault tolerance support that is needed in most CoT scenarios into the system and let the programmer deal with the esoteric needs of the particular scenario.

The JAM is a distributed virtual machine that spans the cloud, fog, and devices. It is hierarchically organized with the root in cloud and leaves at the devices. The fog nodes are in between the cloud and devices. The JAM system and the JAMScript language are designed for a particular fault model of the three-layered CoT, which we show can represent different deployment scenarios.

Instead of designing a new language from scratch, we opted to design JAMScript as a polyglot language that includes C functions and JavaScript functions. The JAMScript specific constructs are responsible for describing the interfaces between C and JavaScript functions. A JAMScript program is compiled into a C component and JavaScript (J) component. For a minimal program execution, one instance of each component should be run. In a typical three-layered CoT execution, C components run in the devices and J components run at the devices, fog and cloud.

This paper makes the following contributions:

- Provides a distributed virtual machine architecture to support a three-layered cloud-fog-device CoT and a fault model that takes disconnections due to device mobility into consideration.
- Presents a design of a polyglot language for CoT that combines two existing languages C and JavaScript and runs on the distributed virtual machine.
- Explains the implementation and discusses the initial experience of deploying toy examples.

This paper is organized as follows. Section II explains the objective of the project and the requirements of CoT

978-1-5090-4607-2/16 $31.00 © 2016 IEEE
DOI 10.1109/CIC.2016.31

programming. The JAM system and the JAMScript language are described in Section III. Implementation details of the proof-of-concept compiler are presented in Section IV. Related work is described in Section V.

II. Objective and Requirements

Our objective is to develop a new programming language and system for CoT that would meet the common programming requirements of CoT deployments. Instead of developing a new language from scratch, we want to take a polyglot approach which will allow us to reuse existing programs and leverage programmer familiarities.

We motivate the CoT programming requirements using problems from several example deployments (e.g., vehicular clouds, smart homes, etc). In vehicular clouds, vehicles supply resources to the cloud and consume services from the cloud. As vehicles move, temporary network partitions can happen in vehicular clouds. Because RPC aims to make remote calls transparent, network partitioning is lumped with server failures and remote application failures. Consequently, from the client application's point-of-view they are indistinguishable. The REST approach, on the other hand, considers fault-tolerance an application specific issue and provides only a basic support. In particular, network partitioning is not effectively handled by either approach.

Because **network partitioning** can occur frequently, we need to distinguish between a service not receiving the request for service and service not responding to the request. We want the system to handle the first scenario and defer the second scenario to the application. That is, the language needs to mask network partitioning so that programmers need not repeat the same procedures across different applications and also the masking can benefit from system-level information that may not be available to the applications.

Consider a smart home that has many devices (e.g., intelligent thermostat) embedded in it to control various functions such as heating, cooling, and humidity control. In many cases, the devices dial into the Internet-based services because of asymmetric network reachability. In a CoT, the asymmetry can extend beyond network reachability. Devices can exercise autonomy and ignore controls from the clouds, hibernate to save power, or move over to another location. This means, when the network link is present, a device should be able to interact with a cloud service; but, the cloud service may not be able to interact with a device even when the network link is up. We refer to this problem as **asymmetric reachability**. The programming language and system need to take this into consideration in its fault model.

CoT can be a very large-scale system where many devices are connected to the cloud and possibly to several fogs. The division of functionality across the different elements need to be carefully organized depending on their capabilities and network location. The programming language could get the guidance of the programmer for the most important aspects of **heterogeneity management**. For instance, in a CoT with cloud, fog, and devices, one of the important issues is the division of functionality across the components.

In CoT, clouds play an important role. They have the "birds-eye-view" of the system to perform important **collective processing** tasks. The data feeding into the collective processing is typically captured at the devices. Therefore, the language and system should support efficient ways of transporting the data to the cloud and collating the data according to a predefined scheme. Once the data is gathered at the cloud, it will be analyzed by data processing routines to extract intelligence and it will be used to control the devices. We need efficient ways of copying the control signals back to the devices from the cloud. In addition to data movement between the devices and cloud, we also need to support grouping function calls. That is, the cloud may need to invoke given functions on a group of devices based on some matching criteria.

III. Language and System for Cloud of Things

A. Overview

Figure 1 shows the overall system architecture for the CoT. The CoT is rooted at the cloud. A device, on the north side, is connected to exactly one fog node or the cloud (not to both). Similarly, a fog node, on the north side, is connected to the cloud. As shown in the figure, the cloud could be connecting to potentially a large number of fogs and devices. Also, a fog could be connecting, on the south side, to large number of devices.

The fog layer can play different roles in the CoT architecture. It can bring cloud services closer to the devices and reduce the latency of access, improve the reliability of the network link (potentially on the same local network), host services relevant in the local context, and improve the scalability of the overall system.

Fogs are created by compute resources that are placed into routers and access points to meet the above mentioned features. Typically, the compute resources run pre-loaded services with cloud backing. This paper treats the fogs as micro IaaS clouds that can be uploaded with application components to have a local service point for the application.

In the case of smart buildings [1], the configuration of the CoT will be fairly static. Maintenance and expansions on the building can trigger configuration changes on the CoT. In the case of vehicular networks, the CoT could be highly dynamic. The placement of the fog nodes itself could be an interesting issue. For instance, fog nodes could be located at the roadside units so that vehicles can receive localized services faster. In this case, the fog is always connected the cloud; but, the vehicle (i.e., device) is not always connected to the fog. Another possibility is to place the fog node in the vehicle (like buses or trains) itself. This would keep the fog node always connected to the devices; however, the cloud-to-fog connection could be unstable.

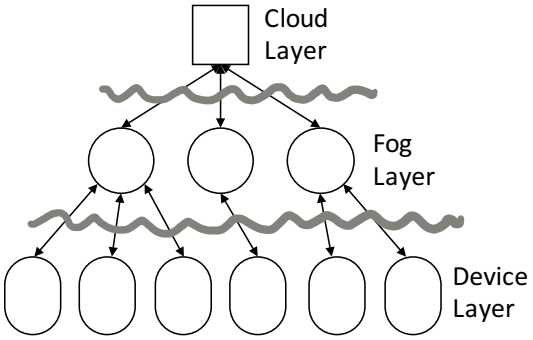

Figure 1: Overall system architecture

B. Fault Model

CoT is a fairly complex and large-scale system that is bound to encounter failures in different ways. The *fault model* presented here enumerates the assumptions we make regarding the failures. Failures can be of two different types: node failures and network failures.

Device nodes in CoT can undergo frequent failures. We assume they fail-stop and do not engage in spurious behavior. Also, the devices can be unresponsive due to scheduled hibernation and disconnection from the CoT. The fog nodes have managed failures because they are part of the network "infrastructure" such as routers and access points. However, introduction of general-purpose computing into these infrastructure elements can create new failure scenarios. We assume that the fog nodes have failure management schemes to ensure very high availabilities. For instance, check pointing, failure detection using a watchdog mechanism, and rapid restarting on failure could be strategies to create highly available fogs. This prevents the fog nodes from being stuck in an unresponsive state. We assume that the cloud nodes are provisioned through an highly available resource management framework such as Apache Marathon so that those nodes are re-spawned as they fail. In addition, an active replication strategy is assumed to be in use to prevent the loss of application state.

Network failures can affect CoT in two ways: disrupt cloud-to-fog connections or disrupt fog-to-device connections. We assume that the fog is either connected to the cloud or to the device; it is never disconnected. This is a key assumption that enables us to claim that the fog can reach the cloud with updates no later than the device.

In a smart building scenario, fog nodes are placed in the building and are part of networking infrastructure. Therefore, if a device such as a thermostat or ventilator is able to reach the cloud, the fog should be able to reach the cloud too. Similarly, in a vehicular networking scenario, the fog nodes should be in the vehicle or at the access point to the Internet (base station or the roadside unit). Given a highly reliable link between the access point and the cloud, the fog should be able to update the cloud no later than the device.

C. The JAM Machine

Figure 2 shows different ways of deploying a JAM machine. The simplest is to run two components, one J and one C, in the same node as shown in Figure 2(a). This method deploys the JAM machine with the highest possible reliability. Figure 2(b) shows a configuration, where a device is directly connected to the cloud without a fog layer. This is suitable for deployments that are tolerant of high and variable latencies (e.g., mobile messaging, sensor networking, etc). Another configuration, where the device just runs the C component is shown in Figure 2(c). The J components runs in a controller that is connected by a highly reliable network to the devices. The sensors and actuators in a car are connected to the electronic control unit using a dedicated bus, which is engineered for high reliability. The JAM deployment on a full three-layered CoT is shown in Figure 2(d).

We define a JAM program as *locally executable* on a node if the C component is able to call the J component and the J component is able to call the C component. Therefore, the JAM program is locally executable in all of the above configurations. We define a JAM program as *globally executable* on a node if the C component is able to call the J components at all levels along the path to the root and the J component at the root is able to call at least one C component and all J components along the path from the root. Similarly, a JAM program is defined as *fog executable* if the C component is able to call the J components up to the fog level and the fog level J component is able to reach at least one C component and all J components along the path.

If a JAM program is globally executable, the machine state remains consistent. Machine state can become *inconsistent* if the JAM program runs when it is not globally executable. In a single node (Figure 2(a)) or controller node (Figure 2(c)), a JAM program always remains globally executable. Therefore, the machine state is always consistent. In other configurations, the JAM program can run while it is locally executable or fog executable. Consequently, the machine state can become inconsistent. The JAM machine expects the application to perform consistency management. The JAMScript language described in the next section has mechanisms to expose the occurrence of inconsistency to the application so it can take corrective measures to resolve it.

In the JAM machine, C to J calls have an *exactly once* reliability semantics. That is, when a C component calls a J component that call is executed exactly once at each J component in the path to the root cloud. Lets consider a device that is offline (i.e., not connected to the fog or cloud). When C calls J, the update is stored in the local device itself. When the device connects to the network, the device

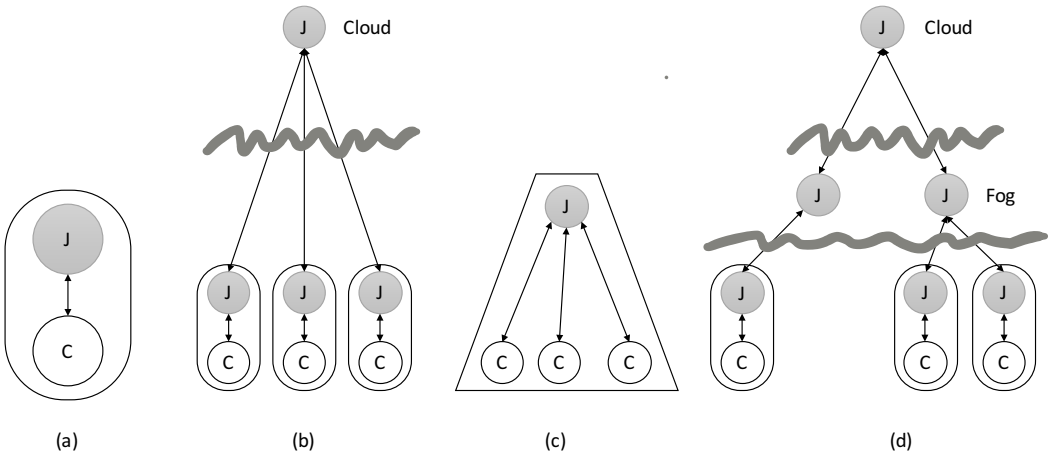

Figure 2: Different modes of deployment of the JAM machine

initiates the updates on the fog and to the cloud. The device is responsible for ensuring that the execution takes place at the fog and cloud.

The J to C calls, on the other hand, have an *at most once* reliability semantics. That is, when a J component make a C call, that call is published to the C nodes. Any J node receiving the published request for execution and at the fog level of the machine would re-publish the request. Any C node receiving the original request because they are directly connected to the cloud or receiving the re-published request from the fog would respond to it by offering to execute the call. No C component would respond twice for the same call of execution. Also, some C components receiving the call may elect not to respond based on the local loading or the unsuitable local settings for executing the function.

The C components in a JAM machine operate independent from one another. That is, the C components that run in the devices follow a distributed memory model (e.g., same variable can have different values at different devices). On the other hand, a JAM machine has a single logical J component that is hierarchically organized in the cloud and the fogs as shown in Figure 1. The JAM machine does *not* provide any consistency management protocols across the different physical realizations of the J component. Instead, the JAM machine leaves the consistency management to the application.

In CoT, devices produce data that needs to be passed on to the cloud for intelligence extraction. The cloud should pass the signals obtained from the data or other sources to the devices for actuation. One way of doing this interchange is to perform J to C and C to J calls. There are many disadvantages for using the function calls for large-scale data interchange.

- *High overhead:* Remote function calls have additional overheads that are not essential for data interchange. We need the most efficient data interchange mechanism that would optimize transfer batch sizes and schedules to maximize throughput or minimize energy usage.
- *Data ordering:* Function calls commit data in the order in which they are executed. In the case of data interchange between the devices and cloud, we want the commit to happen based on production timestamp.
- *Reliability management:* Strict one size fits-all reliability measures can significantly throttle the data throughput.

Figure 3 shows a simplified view of the JAM data buffers (JDB) architecture the JAM machine offers to support high throughput data interchange between the devices and cloud. The JDB puts a high performance data store at each J component. The data store is accessible from each of the C component that is directly connected to the J component. The data store at the cloud J component is accessible to either C components that are connected to the cloud or the data stores in the fog-level J components. The data values written to JDB are immutable. The buffer offered to the C components can be written by multiple writers, whereas the buffer offered to the J component is written by single writer.

D. The JAMScript Language

JAMScript is designed as a *single-threaded* language to create seamless connections and data sharing between devices, cloud and fog. Instead of creating a new language design, JAMScript uses a combination of two well-known programming languages: C code that is run on the devices and JavaScript code that is executed on the cloud/fog. These two components are then integrated into one executable with the necessary glue middleware, which implements the JAM machine, to create a polyglot language. This design allows us to combine the strengths of using each language in the respective environments.

On the device, using C gives us a number of benefits:

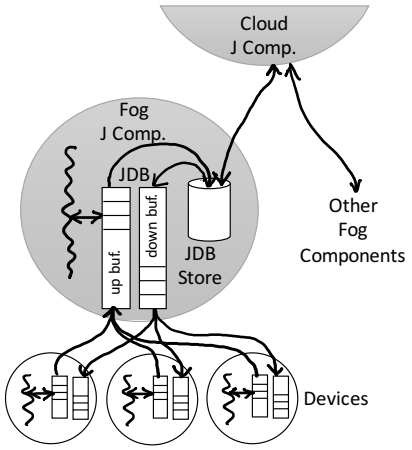

Figure 3: Overview of JAM data buffers

- Code portability: While each different type of device may have its own constraints for development, we can take advantage of the fact that C is a very widely used and supported language in embedded computing. As JAMScript does not alter the host languages, it is possible to run the C component of the JAM executable without many prerequisites.
- Lightweight: C is still lightweight compared to other languages for embedded computing purposes. JAMScript preserves this feature of C.

On the cloud and fog side, using JavaScript gives us the benefits of tapping into the power of modern JavaScript development. JavaScript, running in the Node.js environment, is a rapidly growing language for server side applications. Many new server applications are written in Node.js, and by using JavaScript we create an environment where developers may be more familiar with and can utilize the tools already built into the language to better handle multiple connections as a cloud or fog server.

JAMScript enhances the standard C and JavaScript syntaxes by introducing activities. Activities are similar to functions in the way they are defined and called, but an activity can contain either JavaScript code or C code, independent of the calling language. This feature allows a developer to write an activity containing JavaScript code, that can be called the same way they would call any other C function, but the execution will be done in a JavaScript node. That is, if an activity contains C code then it will be run on the device, and if it contains JavaScript code then it will be run on the fog or cloud. This allows the creation of networked applications just by calling the activity that needs to execute remotely, as if it were a local function. JAMScript activities use a declaration that is written similar to C function headers. This allows the activity code to better integrate into a C environment. By using a C return type and parameter declarations we can make sure that the data being sent into an activity or being returned from an activity will match existing C types.

We introduce two types of activities in JAMScript; synchronous and asynchronous. Synchronous activities look like regular C functions but with the addition of the keyword `jsync` at the beginning of the declaration. A return type must be specified and a value (if not void) will be returned to the calling function as shown in Listing 1. The caller can use this result the same way as the result of any other function, such as assigning the result of the activity to a variable or using it in an expression. Synchronous activity calls are blocking. If there is an error in the remote activity, that error will propagate itself to the local node, potentially halting execution.

```
jsync int power(int a, int b) {
    return Math.pow(a, b);
}

int main() {
    printf("%i\n", power(2, 4));
}
```

Listing 1: Example synchronous activity definition

Asynchronous activities allow non-blocking code to run and are denoted by using the keyword `jasync`. These differ from a regular function, as there is no return type specified. `jasync` activities can only return a handle to the asynchronous call and never a return type, so, the return type is omitted from the declaration as shown in Listing 2. Asynchronous activities can be passed in function callbacks as a way to report back to the local machine that launched the activity. Callbacks are passed into the function by supplying the name of a local function when calling the asynchronous activity. A callback is denoted in the parameters of an activity by using the keyword `jcallback` as the parameter type. The variable of type `jcallback` can then be called like any other function, which when called will then cause the function to execute on the calling node. Multiple callbacks can be passed to a single activity, such as the case of wanting to call one callback if an activity reaches its end correctly, and another callback if there was an error in execution.

```
jasync as_func(int a, jcallback success, jcallback
     error) {
    //Perform a task
    if(taskComplete) {
        success("Result: " + res);
    } else {
        error("Invalid input");
    }
}

void successCallback(char * input);
void errorCallback(char * input);

int main() {
    as_func(1, successCallback, errorCallback);
}
```

Listing 2: Example asynchronous activity definition

In JAMScript, we introduce the concept of 'live' variables which can be used to write into the JDB so that data can be efficiently interchanged with other nodes (cloud, fog and devices). To manage the live variables, JAMScript introduces the `jdata` construct. Live variables can be written to by the devices or cloud/fog nodes. All the live variables are mapped by JAMScript into the global scope. So, no two live variables can have the same name. Listing 3 shows an example `jdata` definition for two variables.

We use three possible `jdata` types to designate the way data flow happens for each variable, they are: logger, broadcaster and shuffler. Writing or reading a `jdata` variable from the C side looks the same as using a regular C variable, while in the JavaScript side a `jdata` variable will be stored as an object and can be accessed the same as any other JavaScript object. The `jdata` variables can only be used within a JAMScript activity, if a user wants to use a `jdata` variable inside a local function they can create a `jsync` activity with code that is the same as language the calling language. This creates a null transform activity that is the same as a normal function.

Logger: Assigning to a variable of type logger is a way of pushing information to the cloud or fog. Variables can only be written to from a C component and can only be read from a J component. Each time a variable is assigned to, its value will be pushed to a stream that is accessible in the J component through an object. The stream can be written to by multiple C components (devices) and the data values will be collated based on the timestamp values. It is possible from the J component to extract a portion of the stream or the entire stream.

Broadcaster: Broadcaster is a way to push values from the cloud/fog to the devices. A broadcaster variable can only be written to from a J component and the value stored in the variable can only be read from the C components.

Shuffler: Shuffler jdata variables are used to send information directly between C components, without passing through a J compoment in between. A shuffler variable can be assigned to and read from a C component, but neither operation is available from the J component. The shuffler is written to using a first-come first-served policy and read using a round-robin policy.

```
jdata {
    int x as logger;
    float b as broadcaster;
}
```

Listing 3: Example jdata definition

In JAMScript, a single program is copied to all nodes (devices, fogs, and cloud) like the single program multiple data model in parallel machines. One exception is that the devices run the whole program (C and J components) and fog/cloud run only the J components. Additionally, depending on the configurations of the devices certain activities defined the program need to be disabled. For example, a thermostat need to only run functions that are related to temperature sensing as shown in Listing 4. To steer the functions to nodes of the suitable types, we can use the `jcond` constructs. This is particularly useful when the J component issues a call for execution. Devices can check the conditions specified by the `jcond` statements to determine whether they should respond to the call.

```
jcond temponly = {
    "type": "thermostat"
}

jsync {temponly} void adjust_temp() {
    // adjust the set temperature value
}
```

Listing 4: Example jcond definition

If a synchronous activity is called from a device, the call fails unless the JAM program is globally executable. That is, if the device is offline and not connected to the fog or the fog is not connected to the cloud, the call is going to fail. An asynchronous activity, on the other hand, can proceed in *parts* (i.e., it can proceed even when the JAM program is locally executable). Therefore, an asynchronous activity can lead to inconsistent machine state. Instead of handling this inconsistency at the machine level, JAMScript exposes such situations to the application and expects the programmer to have provided the code to handle the situation.

We extend the `jcallback` so that instead of passing a single parameter a triple passed into the activity as shown in Listing 5. When the activity completes at the fog level, one of the fog level callbacks (i.e., `succFog` or `errorFog`) will be triggered.

```
jasync as_func(int a, jcallback success, jcallback
     error) {
    //Perform a task
    if(taskComplete) {
        success("Result: " + res);
    } else {
        error("Invalid input");
    }
}

void succDev(char * input);
void errorDev(char * input);
void succFog(char * input);
void errorFog(char * input);
void succCloud(char * input);
void errorCloud(char * input);

int main() {
    as_func(1, [succDev, succFog, succCloud], [
        errorDev, errorFog, errorCloud]);
}
```

Listing 5: Example with triple jcallback

IV. Implementation Details and Status

We have implemented a proof-of-concept compiler for JAMScript using the OMeta/JS compiler-compiler that translates JAMScript source to C and JavaScript and links with

a custom glue middleware (JAMLib) to create a JAMScript executable. The executable created by the compiler can be deployed in modes (a) or (c) in Figure 2. Work is underway to extend JAMLib to support all possible modes of JAM machine deployment.

JAMLib exists in both the C and J components and relies on Nano messages for intercommunication. It leverages the "scalability" communication patterns such as publish-subscribe and survey that are already implemented in nano messages to create a scalable implementation to large number of devices. The messages are binary encoded using the Concise Binary Object Representation (CBOR) before they are sent over Nano sockets. JAMLib is also responsible for supporting event-driven executions in the C side. A user-level threading library is used to implement the event loop with a kernel-level worker thread that services any network calls in a non-blocking manner.

Heartbeat messages are used by JAMLib to reliably detect disconnections between C and J components. Nano messages provide an easy reconnection protocol for the C components when they want to rejoin a previously connected JAM machine.

We developed several toy examples to test the compiler and middleware. First toy example is a simple interpreter that reads a JavaScript program fragment and executes it and prints the output. The second toy example is little complicated, where a simple chat service was implemented. The chat service included a server and client components. Table I shows the line counts of three different implementation of each program. For the simple interpreter we used an embedded JavaScript interpreter called duktape to interpret the JavaScript programs.

Table I: Comparison of the toy example implementations

Program	JAMScript	C	JavaScript
Simple interpreter	22	25	14
Simple chat	98	130	98

It should be noted that fault tolerance was not implemented by any of the implementations. JAMScript has built in support for fault tolerance (not in the current prototype) so the length differential of JAMScript programs would increase when fault tolerance is explicitly coded into pure C or JavaScript.

V. Related Work

In this section we review many related works in IoT-Cloud programming, RPC and fault-tolerance frameworks.

IoT-Cloud programming: We review few programming frameworks among the many that are combining cloud and IoT programming. ELIoT [2] extends the Erlang language for cloud-scale IoT application development. Support for broadcast communication, RESTful interfaces and simulator support are some of the important extensions in ELIoT. Calvin [3] combines the ideas of actor model and flow based computing to merge cloud and IoT. In Calvin, everything is treated as an actor: devices, services, a piece of computation and the cloud. The runtime supports multi-tenancy; migration of actors to different runtimes based on spatial locality and constraints. In the Compose API [4] platform, things are exposed as service objects into the cloud. These objects are exposed to external developers through RESTful APIs. Application developers can develop new applications by composing various service objects through their RESTful APIs. PyoT [5] abstracts wireless sensor networks as software objects which can be composed and manipulated to perform various tasks. Applications can use sensing and actuating capabilities of the motes, shared with the external world through URIs. It provides "in-network processing", wherein an application logic can be directly run on sensor/actuator nodes catering to latency sensitive applications. PatRICIA [6] is a framework which supports IoT application development on cloud platforms. It provides an "intent" based programming model and mechanisms for big data management and analytics in cloud under the IoT domain. Here, the things can be monitored and controlled through tasks (based on intent) from the cloud. Dripcast [7] is a server-less java based application development framework to integrate smart devices into cloud computing infrastructure. This framework can be used for processing and storing java objects in a cloud environment. These remote java objects can be made available on smart things and users can manipulate those objects as if they are local objects. Remote procedure calls on java objects are implemented in a transparent manner.

RPC-frameworks: Finagle [8] developed by Twitter is a protocol agnostic, asynchronous RPC system. Using this developers can write client and server codes in any of the JVM supported languages. Finagle provides support for avoiding TCP connection churn, failure detection of hosts, failover strategies, and load balancing. Apache Thrift [9] developed by Facebook is a framework for implementing RPC with polyglot support. Thrift provides support for type serialization and service implementation. The IDL supports base types, complex types and containers. Transport layer provides data transfer including bidirectional sequenced messaging. BERT-RPC [10] used by github is a transport layer agnostic protocol for performing RPC (synchronous and asynchronous) using BERT as the serialization mechanism. BERT-RPC supports rich set of caching directives like cache expiration and cache validation. gRPC [11] is a RPC protocol developed initially at Google. Using gRPC, applications can call (synchronous and asynchronous) methods on a server application on a different machine written in a different language. It uses "protocol buffers" and exploits advantages of HTTP/2 and TLS. gRPC has its own authentication mechanism that can be used with SSL/TLS.

ICE-E (Internet communication Engine for embedded) [12] is an object oriented middleware platform. Client and servers can be written in different programming languages and can be run on different operating systems and machine architectures. The RPC protocol can use TCP or UDP as the transport layer.

Fault-tolerance frameworks: Atomix [13] is a fault tolerant, event driven framework for distributed systems built on the Raft consistency algorithm [14]. The leader election algorithm handles failures. If a node is unresponsive to heartbeats from the leader, the node will be marked inactive and will be replaced by a standby node. When a leader of the cluster fails, a new election is initiated. On the occurrence of network partition, if the leader is in the side of the quorum, the leader continues to operate normally. If the leader is on the other side of the quorum, then it steps down and servers on the other side of the partition participate in the election of a new leader. Hystrix [15] developed by Netflix, is used as a latency and fault tolerance Java library. It stops cascading of failures, prevents single dependency from using up all container user threads, sheds load and fails fast instead of queuing up requests. It provides support for fall back and user isolation techniques. Akka [16] is a toolkit for developing resilient, scalable distributed applications on JVM and .NET environment. It uses the actor model and for fault tolerance, it uses "Let it Crash" [17] model and allows actors to span over multiple JVMs. Each actor will be the supervisor of its children, and hence the fault handling strategy is defined by the actor itself.

Comparisons with JAMScript: JAMScript has many similarities to the above systems and also several key differences, which we explain below.

JAMScript is designed from scratch for CoT that includes cloud, fog, and devices. A key aspect of this design is a new fault model for CoT that accounts for disconnections between the devices and the infrastructure nodes such as fog/cloud. Compared with the other IoT-cloud programming frameworks, JAMScript explicitly handles a fog layer in two different ways: (a) placement of functions on the fog and (b) facilitating application driven consistency management through callbacks. Also, JAMScript is built on a hierarchical distributed virtual machine that supports two types of data interchange between the devices and the cloud. Reliable low-volume data transfers are handled by the normal RPC calls and unreliable high-volume data transfers are handled by the direct writes to JDBs.

Like the above RPC frameworks, JAMScript allows synchronous and asynchronous procedure calls. Due to the hierarchical nature of the JAM machine, procedure calls north bound from the devices reach to single destination (fog or cloud), whereas the procedure calls south bound from the cloud reaches to potentially large number of destinations some of which may not respond. Unlike the RPC frameworks, JAMScript provides a language layer that is integrated into the two host languages C and JavaScript. Using the compiler we can perform several optimization such as "inlining" to reduce the number of remote calls. Also, the compiler can be used to instrument the JAMScript program to optimize its behavior for performance and fault tolerance.

Compared with fault-tolerance frameworks, JAMScript addresses a very specific fault model in the context of CoT, which we believe is representative of future scenarios. Also, some of the concepts behind these frameworks will complement JAMScript. For instance, we consider cloud and fog nodes to be highly available, which could be realized by leveraging fault-tolerance frameworks such as the above.

VI. Conclusions and Future Work

CoT programming is already recognized by many industry and academic researchers as an important problem that needs specialized language and library support to effectively handle the scale, faults, and diversity that is expected in a realistic deployment. CoT deployments can get significantly complicated from a programming point-of-view due to latency mitigation techniques such as fogs or cloudlets that can be introduced into the overall architecture.

In this paper, we propose a new programming language and system for CoT. The programing language JAMScript is based on a hierarchical distributed virtual machine, which addresses a fault model that takes frequent disconnections due to device mobility into consideration. The language itself merges C and JavaScript in a polyglot manner to create a "distributed systems" programming language.

We implemented a proof-of-concept compiler that can translate JAMScript programs to C and JavaScript and then create JAMScript executables. We have already implemented several toy examples using the JAMScript language and deployed them. We are currently working on extending the underlying middleware to support all features of the language and machine.

JAMScript is designed to address the scale, faults, and diversity we expect with CoT. We believe the best way to evaluate the language features and runtime capabilities towards these objectives is by using emulated backend. With an emulated backend, we can deploy CoTs with varying scales, configurations and introduce faults according to different simulated scenarios. The challenge is to create the backend that will admit JAMScript executables without significant changes.

Acknowledgments

We thank Kininyiruchi Echomgbe for his role in implementing the JAM data buffers support in JAMLib. This research is supported by a Discovery Grant from Natural Sciences and Engineering Research Council of Canada. The source code (not yet publicly released) can be accessed at `http://github.com/anrl/JAM`.

REFERENCES

[1] S. Dawson-Haggerty, A. Krioukov, J. Taneja, S. Karandikar, G. Fierro, N. Kitaev, and D. Culler, "BOSS: building operating system services," in *NSDI'13: 10th USENIX Conference on Networked Systems Design and Implementation*, 2013, pp. 443–457.

[2] A. Sivieri, "ELIoT: A Programming Framework for the Internet of Things," 2014.

[3] P. Persson and O. Angelsmark, "Calvin – Merging Cloud and IoT," *Procedia - Procedia Computer Science*, vol. 52, pp. 210–217, 2015.

[4] J. L. Pérez, Á. Villalba, D. Carrera, I. Larizgoitia, and V. Trifa, "The COMPOSE API for the internet of things." *23rd International Conference on World Wide Web Companion*, pp. 971–976, 2014.

[5] A. Azzara, D. Alessandrelli, M. Petracca, and P. Pagano, "Demonstration abstract: PyoT, a macroprogramming framework for the IoT." *IPSN*, pp. 315–316, 2014.

[6] S. Nastic, S. Sehic, M. Vögler, H.-L. Truong, and S. Dustdar, "PatRICIA – A Novel Programming Model for IoT Applications on Cloud Platforms," in *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, 2013, pp. 53–60.

[7] I. Nakagawa, M. Hiji, and H. Esaki, "Dripcast - Architecture and Implementation of Server-less Java Programming Framework for Billions of IoT Devices," *Journal of Information Processing*, vol. 23, no. 4, pp. 458–464, 2015.

[8] "Finagle," https://twitter.github.io/finagle/guide/index.html, accessed: 10-July 2016.

[9] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," *Facebook White Paper*, vol. 5, no. 8, 2007.

[10] "BERT-rpc," http://bert-rpc.org/, accessed: 10-July 2016.

[11] "gRPC," http://www.grpc.io/docs/guides/concepts.html, accessed: 10-July 2016.

[12] "Ice-e," https://zeroc.com/products/ice-e, accessed: 10-July 2016.

[13] "Atomix," http://atomix.io/atomix/docs/, accessed: 10-July 2016.

[14] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 305–319.

[15] "Hystrix," https://github.com/Netflix/Hystrix/wikiprinciples, accessed: 10-July 2016.

[16] "Akka," http://doc.akka.io/docs/akka/2.4/intro/what-is-akka.html, accessed: 10-July 2016.

[17] J. Armstrong, "Making reliable distributed systems in the presence of software errors," Ph.D. dissertation, The Royal Institute of Technology Stockholm, Sweden, 2003.