ThingsJS: Towards a Flexible and Self-Adaptable Middleware for Dynamic and Heterogeneous IoT Environments

Julien Gascon-Samson
University of British Columbia
Vancouver, BC, Canada
julien.gascon-samson@ece.ubc.ca

Mohammad Rafiuzzaman
University of British Columbia
Vancouver, BC, Canada
rafiuzzaman@ece.ubc.ca

Karthik Pattabiraman
University of British Columbia
Vancouver, BC, Canada
karthikp@ece.ubc.ca

ABSTRACT

The Internet of Things (IoT) has gained wide popularity both in academic and industrial contexts. Nowadays, such systems exhibit many important challenges across many dimensions. In this work, we propose *ThingsJS*, a rich Javascript-based middleware platform and runtime environment that abstracts the inherent complexity of such systems by providing a high-level framework for IoT system developers, built over Javascript. ThingsJS abstracts several large-scale distributed systems considerations, such as scheduling, monitoring and self-adaptation, by means of a rich constraint model, a multi-dimensional resource prediction approach and a SMT-based scheduler to properly schedule and manage the execution of high-level, large-scale distributed applications on heterogeneous physical IoT devices. ThingsJS also provides a rich inter-device communication framework built on top of the widely-used publish/subscribe/MQTT paradigm. Finally, ThingsJS also proposes a rich inter-device Javascript-based code migration framework to support the transparent migration of live IoT components between heterogeneous devices.

CCS CONCEPTS

• **Software and its engineering → Development frameworks and environments**; **Runtime environments**; *System description languages*;

KEYWORDS

IoT, Internet of Things, Javascript, Scheduling, Code Migration, Publish/Subscribe, MQTT, Dependability, Security

ACM Reference Format:
Julien Gascon-Samson, Mohammad Rafiuzzaman, and Karthik Pattabiraman. 2017. ThingsJS: Towards a Flexible and Self-Adaptable Middleware for Dynamic and Heterogeneous IoT Environments. In *M4IoT'17: M4IoT'17: Middleware and Applications for the Internet of Things, December 11–15, 2017, Las Vegas, NV, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3152141.3152391

M4IoT'17, December 11–15, 2017, Las Vegas, NV, USA

ACM ISBN 978-1-4503-5170-6/17/12…$15.00
https://doi.org/10.1145/3152141.3152391

1 INTRODUCTION

The Internet of Things (IoT) involves multiple devices across many domains that are inter-connected in order to provide and exchange data. IoT systems exhibit important challenges across many dimensions, such as very high device heterogeneity, and the usage of many diverse programming languages, APIs and protocols. These factors, combined with the large-scale nature of many IoT systems which can often comprise several thousands of nodes, require developers to handle complex distributed systems and software dependability considerations. These can in turn open the door to bugs and security vulnerabilities [7].

In this paper, we propose *ThingsJS*, a rich Javascript-based middleware and runtime environment aimed at addressing the above challenges. While JavaScript has gained wide popularity as a programming language for web applications [13, 16], it has recently been proposed as a language for IoT programming as well. This is because of its event-driven nature, and large installed base of libraries and developers who know the language. Consequently, there have been a number of open-source Virtual Machines (VMs) developed for running JavaScript code on IoT devices [2, 3, 11, 23].

This paper presents a holistic view of our ThingsJS platform, the set of challenges that it aims at solving, as well as the set of research contributions. We note that at this stage, we have neither formalized all our design choices nor conducted formal evaluations of our system; rather, this paper outlines our main goals and design choices, as well as relevant related work. We provide the following main contributions in ThingsJS:

- A platform comprising a set of APIs and a set of high-level programming and MQTT-based communication paradigms to efficiently write the code for the various components of an IoT system in JavaScript, as well as a declarative syntax to describe the relationships and constraints between the components and devices (Section 2).
- A flexible model where the components are scheduled and dispatched to IoT devices *on-the-fly*, while ensuring that constraints are respected, by leveraging a novel machine learning-based resource prediction model combined with an SMT solver-based scheduler (Section 3).
- A state-preserving high-level JavaScript migration engine that transparently migrates the execution of JavaScript-based applications across heterogeneous IoT devices (Section 4).

2 SYSTEM ARCHITECTURE

The high-level architecture of the ThingsJS Framework is made of several distributed components and is presented in Figure 1. From a holistic point of view, a ThingsJS environment comprises a

highly-distributed *ThingsJS Application* (Section 2.1), and dynamically manages its execution over a set of heterogeneous *devices* through the *ThingsJS Framework* (Section 2.2).

2.1 ThingsJS Application

A ThingsJS application is made of several components and is executed in a distributed fashion across the various devices of a ThingsJS system. Note that in our model, a device can refer to either an IoT device, or other *processing* nodes, such as a cloud VM.

2.1.1 Source Code. At its core, a ThingsJS application contains source code expressed in a high-level language, in a modular fashion (i.e., in the form a set of components). In this paper, the language we consider is JavaScript, due to its widespread popularity and portability across a wide range of devices [2, 11]. The use of a high-level language also allows developers to shield themselves from platform-specific considerations. As such, as part of the ThingsJS framework, we provide a set of core APIs that developers can consume to write their applications. Such APIs include abstractions of file system access (as the components might run on arbitrary nodes), hardware access (i.e., sensors), publish/subscribe for inter-component communications (Section 2.3) and are designed to be *migration-aware*, so that they can properly sustain a code migration (Section 4). Developers are free to use external libraries as in regular JavaScript, but they are responsible for ensuring that such libraries can be migrated as well.

2.1.2 Components. As part of the ThingsJS philosophy, developers are encouraged to modularize their source code as much as possible, following the software engineering principle of one component ↔ one responsibility, and let the runtime dynamically decide on the best placement of components to devices. In this spirit, having a set of small components allows for greater scheduling flexibility, and partially shields the developer from scheduling considerations. In turn, this allows the use of more elaborate heuristics depending on the desired optimization criteria. Implementation-wise, in Javascript, one component would typically correspond to one `js` file. The example outlined in Figure 1 models a rudimentary ThingsJS application designed to manage the temperature inside the various rooms of a given building. It models three components: (1) a sensor, which is in charge of collecting readings from temperature sensors located in the different rooms a the building; (2) an actuator, which allows for regulating the power output to adjust the temperature across the various rooms and (3) a regulator, which receives temperature sensor readings from the sensors and which instructs actuators to increase or decrease the power output. In this example, we identified three roles/responsibilities, which led us to decouple our implementation into three components, and let the ThingsJS Scheduler schedule the execution of the components.

2.1.3 Device and Component Declaration. In addition to the source code in the form of `js` files, developers must specify a set of devices on which the components will be run. Also, as there will likely be more than one instance of some of the components (i.e., a building can contain many temperature sensors and actuators, but perhaps only one regulator), then developers must also specify the different instances of each component to be executed. Note that ThingsJS allows for these to dynamically change, as devices and components can dynamically be added and/or removed. In the example outlined in Figure 1, two IoT devices are defined: `smart3js1`, (e.g., a Raspberry PI Model 3b), and `smart0js1`, (e.g., a Raspberry PI Model 0W). In addition, a cloud VM device is defined (`cloud1`).

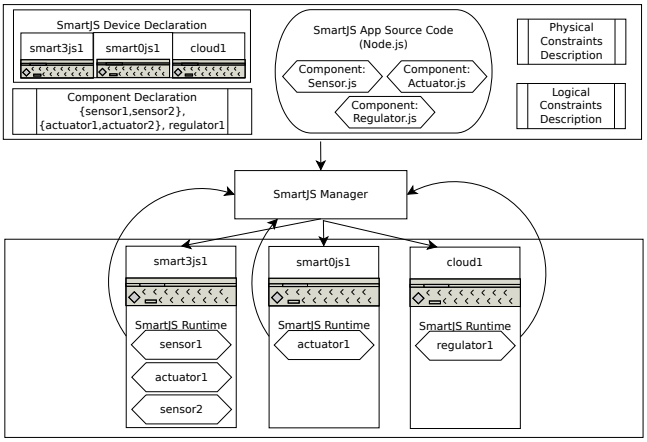

Figure 1: High-Level Architecture of ThingsJS

2.1.4 Physical and Logical Constraints. As we mentioned, we let the ThingsJS Manager schedule the placement of components to devices. In order for the *Manager* to make optimal choices, a set of constraints must be specified, as the set of devices which can hold a given component instances might be restricted. For instance, a sensor component instance might have to be restricted to a specific device or set of devices, due to the presence of the *physical* sensor. On the other end, while there might be some flexibility allowed for other components, the component may be too demanding in terms of system resources for it to be run on a given device. For example, the `regulator` component might be too demanding to be run on a Raspberry PI Model 0W. ThingsJS proposes mechanisms to express such constraints, and distinguishes between two sets of constraints:

Physical Constraints which model the system characteristics of the various ThingsJS *devices* themselves. They typically model the physical capabilities (e.g., CPU, memory), the networking capabilities (e.g., incoming and outgoing bandwidth and latency), the power source (e.g., battery or grid). Note that some of these constraints are periodically refreshed as measurements are performed.

Logical Constraints, which model the characteristics of the ThingsJS *components*. They model, for each component, the various resource requirements (CPU, memory, incoming/outgoing bandwidth and latency), and can also be used to bind specific components to specific devices, e.g., due to dependencies with hardware.

2.2 ThingsJS Framework

The ThingsJS Framework includes infrastructure components that are responsible for managing the dynamic execution of a ThingsJS application over the set of available IoT devices.

2.2.1 Manager. The ThingsJS Manager manages the execution of all components across all devices. It takes as input a ThingsJS application as described in Section 2.1 and schedules and monitors its

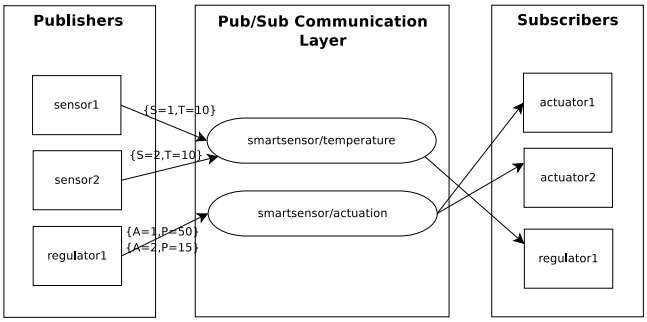

Figure 2: Topic-Based Publish/Subscribe Model

distributed execution across all participating ThingsJS devices. The deployment process itself is currently done through a command-line interface. We are however working on an API and a web interface to ease this process. Note that the scheduling is done through the *ThingsJS Scheduler* (Section 3.2).

2.2.2 Runtime. An instance of the ThingsJS Runtime is present on every device. It locally manages the execution of all components on the device. It also includes a *collector* component which monitors available system resources and captures highly detailed performance data for each component to be fed to the Scheduler's prediction model for scheduling purposes (Section 3).

2.3 Inter-Component Communications

In *ThingsJS*, we require that all inter-component communications follow a topic-based publish/subscribe [9] (MQTT) model. The choice of this model was primarily motivated by the logical decoupling of content producers from content consumers that it provides, which allows for abstracting many networking-related considerations such as the management of ip addresses/ports, low-level connections, protocols, etc. Also, due to its lightweightness and simple yet flexible conceptual model, the use of topic-based publish/subscribe enjoys widespread popularity in IoT applications. Requiring all components to use pub/sub also allows ThingsJS to optimize the pub/sub overlay at the infrastructure level, while abstracting such considerations from ThingsJS Application developers, i.e., they only need to know that a pub/sub service is always available and consume the appropriate APIs that are provided, irrespective of *how* the service is provided.

From an implementation standpoint, a topic-based pub/sub infrastructure can be provided in several different ways: under a client-server model [1], as a service in the cloud [10], and also in a peer-to-peer fashion [6, 21]. In the context of this paper, we opted for a classic client-server approach due to its simplicity, but we do plan as future work to explore alternative flexible approaches at providing pub/sub in an adaptive way.

Figure 2 shows an example usage scenario in the context of our *SmartTemperature* demo app. On the left, `sensors` 1 and 2 act as publishers and transmit relevant publication messages (sensor data in the form of a sensor ID and a temperature intensity measurement) to topic `smartsensor/temperature`. The `regulator` component is the sole subscriber to that topic. In response, the `regulator` produces actuation publications (actuator ID, power output) published

```
1  // ...
2
3  // Connect
4  pubsub.connect(function() {
5
6      // Subscribe to temperature messages
7      pubsub.subscribe("smartsensor/temperature", function(d) {
8
9          if (d.temperature > threshold) {
10             pubsub.publish("smartsensor/actuation", {
11                 id: d.id,
12                 powerVariation: -5
13             });
14         } else if (d.temperature < threshold) {
15             pubsub.publish("smartsensor/actuation", {
16                 id: d.id,
17                 powerVariation: 5
18             });
19         }
20
21     });
22 });
```

Figure 3: Regulator - Sample Code

over topic `smartsensor/actuation`, which are then consumed by the `actuator` components to regulate the power output. Figure 3 illustrates the sample pub/sub-related code for the *SmartTemperature* demo application, based on the model shown in Figure 2.

3 DYNAMIC SCHEDULING

Considering a set of components and a set of devices, the *ThingsJS Scheduler* (a subcomponent of the *Manager*) comes up with an optimal way to schedule the execution of every component on to a specific IoT device, while respecting the physical and logical constraints described in Section 2.1.4. In other words, for every component, the Scheduler has to pick the best target device among all (potentially thousands of) available IoT devices.

3.1 Optimal Device for a given Component

We consider a device D_i to be optimal for component C_i if it minimizes the execution time of C_i, while respecting all constraints and accounting for the current execution of all other components on D_i. In future work, we plan on supporting other optimization criteria; e.g., minimizing component-to-component response time, bandwidth usage, costs, etc., as different components may have different optimization goals.

3.2 ThingsJS Scheduler: Challenges

Selecting the best device to host a given component is challenging due to the following reasons:

(1) Non-Deterministic Performance. While the execution time of a component C_i varies according to the resources in a given device D_i, there can often be some noise. In other words, differences in results can be observed upon running the same component C_1 under similar system conditions.

(2) IoT Devices Heterogeneity. The ideal IoT device choice to host a given component is application-dependent, as different applications will exhibit different resource requirements (i.e., CPU, RAM, bandwidth, etc.).

(3) Computational Overhead and System Complexity. Establishing a strong correlation between the set of available resources on a given platform and the set of execution times of a given program is a challenging problem, due in part to (2), but also due to the

fact that several distributed and operating system-related factors come into play. Extracting the usage of various resources from all IoT devices in order to estimate the execution time of different applications incurs high computational overhead and complexity. There is a need to develop an efficient resource prediction model, which is one of our key contributions.

3.3 ThingsJS Scheduler: Design

The *ThingsJS Scheduler* operates in two steps. First, it uses Machine Learning (ML) techniques to predict the execution time of a given component over all available devices $D_i \in D$. Then, it makes use of a SMT-based (Satisfiability-Modulo-Theory) solver to determine the best assignation of components to devices which minimizes execution time and which respects the set of physical and logical constraints, as described in Section 2.1.4.

3.3.1 Predicting Execution Times. As mentioned previously, each component periodically reports its performance metrics (usage of a wide range of system resources) to the *Collector* subcomponent of the ThingsJS Manager. In the training phase, the ThingsJS Scheduler continuously feeds the data received from the *Collector* to the prediction model so that the latter is continuously refined with live data (i.e., available system resources) from all available devices. Then, with sufficient training data, and by making use of several different regression models [15, 20, 22, 22, 24], the prediction model can be used to predict what the execution time of an arbitrary component C_i would be over the set of devices $D_i \in D$ with high accuracy. Note that each regression model here serves its own purpose to predict the execution times of different kinds of components, and their corresponding hyper-parameters are dynamically tuned each time based on the nature of the input data to give the best possible prediction. The final prediction is generated by processing the prediction results from all of them. The training process is iterative and follows a feedback loop: as scheduled components are executed over devices, their observed performance metrics are collected and continuously fed back to the prediction model to improve it.

3.3.2 Scheduling. The predicted set of execution times for component C_i over the set of IoT devices $D_i \in D$ are fed to the ThingsJS Scheduler, which, as mentioned previously, is initially responsible for determining where each component should be placed. To that end, it uses a SMT solver (i.e., Z3 [8]) to find the most suitable *global* configuration in an efficient manner. Note that since the scheduling occurs at the macro level, one cannot necessarily assume that a given component C_i will be placed on the device which *minimizes* its execution time, as the scheduler must come up with the best global configuration while respecting the constraints.

3.3.3 Adaptability. As the system is running and conditions change (e.g., constraints become violated, new component instances are launched, some nodes switch to battery power), there may be a need for migrating already-executing components to different devices (see Section 4). In the case of constraints violations, the scheduler must come up with an alternate configuration to resolve the reported violations. If new components are being introduced, then the scheduler must also come up with a suitable configuration to dispatch the execution of these components. In other cases, the scheduler can produce an alternate configuration which would minimize the objective function for some of the components by means of a better placement. In all these cases, the *Scheduler* must also take into consideration the *costs* of migrating components, in order to find the right balance between migrating components and the costs involved in the migration. The exact formulation of the cost model is deferred to future work.

3.3.4 Fault Tolerance. Both IoT nodes and components are subject to arbitrary failures. If ThingsJS components exceed the available resources (e.g., memory), they are prone to failure. Upon a component crashing, the ThingsJS Runtime on the appropriate IoT node will detect the failure, inform the Master who will then re-execute the Scheduler to reschedule that component on another node. We can extend the execution time prediction model in 3.3.1 to predicting the failures of ThingsJS components on IoT nodes before they occur, in order to preemptively reschedule components. In the case of a node failure, the ThingsJS Manager will detect the failure (loss of communication with the appropriate Runtime), and will trigger the rescheduling of all components that were executing on that device, following the same approach.

The code migration approach that we discuss next (Section 4) can be used for fault tolerance as well. We plan on working on automatic snapshotting of the state of ThingsJS components so that their latest known state can be restored upon a failure occuring.

4 CODE MIGRATION

Upon the *SmartJS Manager* generating a new configuration in which some of the component instances are moved to a different device, it becomes necessary to dynamically *move* the execution of such components from the *old* device to the *new* one. Upon component C_1 being moved from device D_1 to D_2, a trivial approach would simply consist of the *Manager* asking the *Runtime* on D_1 to shut down component C_1 and asking the *Runtime* on D_2 to start a new instance of C_1 on D_2. Obviously, this approach is very limited, as the current state of C_1 will not be preserved. While some IoT components might be stateless, such as simple sensors, we envision that as IoT devices become more and more powerful, such devices will be executing more complex applications. For instance, taking the example shown in Figure 2, one could imagine a sensor or actuator component that could remember a set of local historical values and perform some more elaborate local computations based on such values. There are also already some discussions in executing machine learning applications on IoT nodes [14].

For stateful Javascript IoT applications, a simple migration approach could be, for all components, to provide APIs allowing them to save and restore their current state. Such a solution would however require ThingsJS developers to implement the serializing behavior for all components, which would be cumbersome. Therefore, there is a compelling need to come up with an automated and dynamic migration approach that will alleviate ThingsJS developers from such considerations.

4.1 Process Migration: Challenges

Process migration techniques can be used to dynamically migrate the execution of a running process from one device to another. Many techniques have been proposed [17, 19, 25, 26], which typically operate by saving and restoring the process memory space.

In our case, components are written in JavaScript, a high-level language. Process migration would then involve migrating the virtual machine (VM) process itself, which is in charge of executing a given component, at the expense of potentially large serialized states and high overheads on resource-constrained IoT devices. Serializing the VM process would also prevent the ThingsJS Runtime from executing multiple JavaScript components within the same VM space - which might be desirable to reduce the memory footprint of each component - as the serialization of the VM process would serialize all components managed by the VM.

In addition, considering that an IoT system will most likely feature a set of heterogeneous devices and that the ThingsJS runtime might schedule the execution of a given component on different physical devices, transparent process migration techniques are infeasible due to low-level platform differences (e.g., processor, memory-space, etc.) between them. Therefore, we do not plan to take this approach.

4.2 JavaScript Component Migration

As an alternative to process migration, we propose, as a novel contribution, a set of techniques to perform live migration of JavaScript components (applications) from one VM to another, irrespective of the physical platform. Provided that the migration occurs purely in the JavaScript application space, such an approach allows us to solve the challenges outlined above: supporting the execution of several ThingsJS components in the same VM context, and achieving component migration across heterogeneous devices. The following subsections give an insight into the main code migration-related challenges that we are currently working on solving, as well as an insight into the solutions that we are proposing.

4.2.1 Main Challenge: Handling Closures. At its core, a JavaScript application comprises a *global* scope in which all the variables and functions are defined by default. In order to capture the state of a JavaScript application, one needs to be able to *recursively* capture the state of all variables defined in the global scope. JavaScript also exposes a rich object model, in which properties can be dynamically added to objects, which, combined with a rich reflection API, allows one to recursively iterate through objects to serialize their state.

However, JavaScript considers functions as data, which can lead to increased complexity, as functions can be anonymously defined, bound to variables and returned. Increased complexity stems from the ability of functions to not only access the variables defined in their own *scope*, but also those defined within their ancestors' scope (closures). The code sample illustrated in figure 4 illustrates the use of closures in Javascript to define a *Counter* entity. In this code sample, the `Counter` function stores the initial value of the counter, and returns another (anonymous) function which increments the counter. At line 5, we observe that the *nested* function can access and modify variable `value` defined in the parent scope. Then, upon storing the return value of `Counter` in a variable, one can invoke the anonymous functions (lines 14 and 15) to increment the counter.

Saving closures brings additional complexity, stemming from the fact that in order to properly store the state of all variables which point to functions (i.e., `f` and `g`), one needs to store the state of the closures that such functions contain (the *scope*, which refers to all variables that the function can access). In the case of `f` and `g`, the scope would contain the current value of variables `value` and `val`, private to each *instance* of the closure.

```
1  function Counter(val) {
2     var value = val;
3
4     return function() {
5        value = value + 1;
6        // Can access parent function local variable
7        return value;
8     }
9  };
10
11 var f = Counter(5);
12 var g = Counter(2);
13
14 document.writeln( f() ); // Prints 6
15 document.writeln( g() ); // Prints 3
```

Figure 4: Counter JavaScript Example

From a graph theoretic perspective, the hierarchy of closure scopes form a tree-like structure [13] that one has to walk to properly save the hierarchical state of all closures defined in a given JavaScript application. However, the JavaScript reflection API does not contain provisions for accessing the scope of closures, i.e., the set of variables that a given function defines or can access.

4.2.2 Saving and Restoring Closures. In order to solve the serialization challenges described at the previous section, [13] et al. propose an approach in which the Javascript Virtual Machine (VM) is instrumented to allow access to the internal states, in order to be able to track closures scopes and to properly serialize them. Then, reconstruction code is generated to be able to reconstruct the program context. If possible, we prefer to avoid VM instrumentation, as such techniques might be VM and platform-dependant.

As part of our contributions, we come up with an approach in which we instrument the code prior to it's execution by means of static Javascript code injection, with the goal of exposing the data hierarchy and closure nesting (i.e., the *scope tree*) within the program under a dynamic tree-like structure. Upon a *snapshot* being taken, the dynamic scope tree is walked downwards starting from the top to be serialized. The restoration process then consists of parsing the state and generating appropriate closure reconstruction code, which yields a program that is functionally equivalent to the original program at the moment of taking its snapshot. In other words, the code is different, but the data and the logic that it exposes are equivalent, and execution can then proceed.

4.2.3 Additional Challenges. An additional challenge is when to perform the migration in a JavaScript program. As we cannot easily access, serialize and deserialize the JavaScript call stack and event queue, we *schedule* the saving of the state to be executed at the end of the event queue by means of timers (since JavaScript is single-threaded, it does not allow the current thread to be preempted). We believe however that this is an acceptable solution due to the event-based, asynchronous nature of Javascript. We are also looking at supporting the serialization of nested libraries. Another important challenge that we aim at tackling is the correct serialization of the MQTT/pub-sub layer, which will involve serializing the set of all

subscribers-topics, and ensuring that no publications are lost during the migration process.

5 RELATED WORK

To the best of our knowledge, the idea of scheduling applications on IoT nodes is novel, and we think that ThingsJS opens the door to important contributions in this area. There has been some prior work in scheduling applications in a cloud setting [5, 12], but we think that using such approaches directly could be impractical due to the many key differences between cloud and IoT systems.

There has been some previous work in migrating Javascript code across browsers, but to the best our knowledge, we found no approach that dealt with the specific problem of migrating Node.js out-of-browser applications. Most relevant related work include [16], which proposes a framework to migrate the state of browser-based Javascript applications across web browsers, and [13], builds upon the prior approach by improving the handling of nested closures by instrumenting the VM to access its internal state. Note that our approach improves on [13] by avoiding low-level VM instrumentation in order to increase cross-device/VM portability.

From a systems perspective, there has been some work in proposing virtual machines and frameworks for developing and running IoT applications. Garvin et al. [11] introduce the Samsung IoT.js project, an embedded, tiny JavaScript virtual machine aiming at running JS code on resource-constrained IoT devices. On the other hand, the Intel XDK framework [2] aims at proposing a framework for writing mobile and IoT applications using web technologies (HTML5, Javascript). Some recent research and industry proposals aim at proposing emergent paradigms that can be used to model the roles and relationships among IoT systems, such as actor-based approaches, which can be leveraged to model the communication flows between components [4, 18]. We note however that ThingsJS differ by providing an integrated platform that aims at solving challenges across multiple dimensions.

6 CONCLUSION

In this paper, we presented an overview of ThingsJS, our rich Javascript-based IoT middleware which aims at simplifying the development of IoT applications across heterogeneous IoT devices. We discussed the holistic architecture and the technical decisions of ThingsJS, as well as the main contributions that we intend to bring, across several domains. Our contributions are (1) providing a rich platform that can abstract scheduling, communications and fault tolerance considerations, (2) predicting the load of IoT applications and dynamically scheduling their placement across devices, and (3) enabling the transparent migration of rich IoT JavaScript applications across heterogeneous IoT devices.

ACKNOWLEDGMENTS

This work is supported by a research gift form Intel, a Discovery grant, and a Post-Doctoral Fellowship from the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] 2013. Redis Website. (2013). http://www.redis.io/
[2] 2014. Intel XDK. (2014). https://software.intel.com/en-us/xdk
[3] 2017. mJS. (2017). https://github.com/cesanta/mjs
[4] 2017. Node-RED Website. (2017). https://nodered.org/
[5] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics.. In *NSDI*. 469–482.
[6] M. Castro, P. Druschel, A.-M. Kermarrec, and A.I.T. Rowstron. 2002. Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications* 20, 8 (2002), 1489–1499.
[7] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. 2011. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *Proceedings of the 20th USENIX Conference on Security*. USENIX, Berkeley, USA.
[8] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766
[9] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.* 35, 2 (2003), 114–131. https://doi.org/10.1145/857076.857078
[10] J. Gascon-Samson, F. P. Garcia, B. Kemme, and J. Kienzle. 2015. Dynamoth: A Scalable Pub/Sub Middleware for Latency-Constrained Applications in the Cloud. In *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*. 486–496. https://doi.org/10.1109/ICDCS.2015.56
[11] Evgeny Gavrin, Sung-Jae Lee, Ruben Ayrapetyan, and Andrey Shitov. 2015. Ultra Lightweight JavaScript Engine for Internet of Things. In *SPLASH Companion 2015*. ACM, New York, NY, USA, 19–20.
[12] Herodotos Herodotou, Fei Dong, and Shivnath Babu. 2011. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 18.
[13] Jin-woo Kwon and Soo-Mook Moon. 2017. Web Application Migration with Closure Reconstruction. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. Geneva, Switzerland, 133–142.
[14] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. 2016. DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. 1–12. https://doi.org/10.1109/IPSN.2016.7460664
[15] Andy Liaw, Matthew Wiener, et al. 2002. Classification and regression by randomForest. *R news* 2, 3 (2002), 18–22.
[16] James Teng Kin Lo, Eric Wohlstadter, and Ali Mesbah. 2013. Imagen: Runtime Migration of Browser Sessions for Javascript Web Applications. In *Proceedings of the 22Nd International Conference on World Wide Web (WWW '13)*. ACM, New York, NY, USA, 815–826.
[17] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. 2000. Process Migration. *ACM Comput. Surv.* 32, 3 (Sept. 2000), 241–299. https://doi.org/10.1145/367701.367728
[18] Per Persson and Ola Angelsmark. 2015. Calvin–merging cloud and iot. *Procedia Computer Science* 52 (2015), 210–217.
[19] E. T. Roush and R. H. Campbell. 1996. Fast dynamic process migration. In *Proceedings of 16th International Conference on Distributed Computing Systems*. 637–645. https://doi.org/10.1109/ICDCS.1996.508015
[20] Peter J Rousseeuw and Annick M Leroy. 2005. *Robust regression and outlier detection*. Vol. 589. John wiley & sons.
[21] Vinay Setty, Maarten Van Steen, Roman Vitenberg, and Spyros Voulgaris. 2012. PolderCast: Fast, Robust, and Scalable Architecture for P2P Topic-based Pub/Sub. In *International Middleware Conference (Middleware)*. 271–291.
[22] Donald F Specht. 1991. A general regression neural network. *IEEE transactions on neural networks* 2, 6 (1991), 568–576.
[23] S. Tilkov and S. Vinoski. 2010. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing* 14, 6 (Nov 2010), 80–83. https://doi.org/10.1109/MIC.2010.145
[24] Hrishikesh D Vinod. 1978. A survey of ridge regression and related techniques for improvements over ordinary least squares. *The Review of Economics and Statistics* (1978), 121–131.
[25] Amirreza Zarrabi. 2012. A generic process migration algorithm. *International Journal of Distributed and Parallel Systems* 3, 5 (2012), 29.
[26] S. Zhongyuan, Q. Jianzhong, L. Shukuan, and Z. Qiang. 2015. Use Pre-record Algorithm to Improve Process Migration Efficiency. In *2015 14th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*. 50–53. https://doi.org/10.1109/DCABES.2015.20