

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

Julien Gascon-Samson
University of British Columbia
Electrical and Computer Engineering
Vancouver, BC, Canada V6T 1Z4
julien.gascon-samson@ece.ubc.ca

Mohammad Rafiuzzaman
University of British Columbia
Electrical and Computer Engineering
Vancouver, BC, Canada V6T 1Z4
rafiuzzaman@ece.ubc.ca

Karthik Pattabiraman
University of British Columbia
Electrical and Computer Engineering
Vancouver, BC, Canada V6T 1Z4
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 SAT-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 *Proceedings of the 4th Workshop on Middleware for Context-Aware Applications in the IoT, Las Vegas, Nevada USA, December 2017 (MAIOT 2017)*, 6 pages. https://doi.org/10.475/123_4

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

MAIOT 2017, December 2017, Las Vegas, Nevada USA

© 2017 Copyright held by the owner/author(s).

ACM ISBN 123-4567-24-567/08/06...\$15.00

https://doi.org/10.475/123_4

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, render integration difficult, and also require developers to handle complex distributed systems and software dependability considerations. These can in turn open the door to bugs and security vulnerabilities [6].

In this paper, we propose *ThingsJS*, a rich Javascript-based middleware and runtime environment aimed at solving the above challenges. While JavaScript has gained wide popularity as a programming language for web applications [12, 14], it has also been proposed as a viable language for IoT. 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 Virtual Machines (VMs) developed for running JavaScript code on IoT devices, and many have been released as open-source [2, 10].

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 not yet formalized all our design choices nor conducted formal evaluations of our system; rather, this paper presents our main orientations 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 that can be leveraged 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 a SAT-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* (discussed in the next section) and dynamically manages its execution over a set of heterogeneous *devices* through the *ThingsJS Framework*, which we describe at 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 a any IoT device, but it is not restricted to this definition, as a device might also refer to other “processing” nodes, such as a cloud VM.

2.1.1 Source Code. At it’s 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 the context of this paper, we opted for Javascript, due to it’s widespread popularity and that it can be run across a wide range of devices [2, 10]. The use of a high-level language also allows developers to abstract 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 component might run on arbitrary nodes), hardware access (i.e., sensors), publish/subscribe for inter-component communications (more in section 2.3) and are designed to be *migration-aware*, so that they can properly sustain a code migration (more on code migration in section 4). Developers are free to use external libraries.

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, allowing 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 across the devices.

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

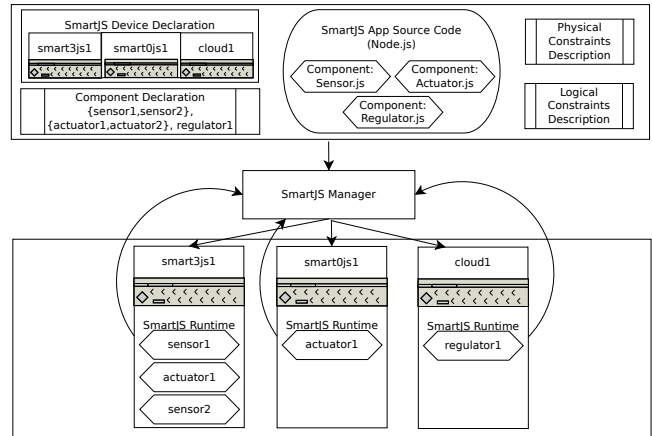


Figure 1: High-Level Architecture of ThingsJS

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, which models a Raspberry PI Model 3b, and smart0js1, which models a Raspberry PI Model 0W. In addition, a cloud VM device is defined (cloud1).

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 only, due to the presence of the *physical* sensor.

On the other end, while there might be some flexibility allowed for a given component if it is not bound to the physical world, it might be the case that the component is too demanding in terms of system resources for it to be run on a given device. In the present context, for instance, 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 (i.e., CPU, memory) as well as the networking capabilities (i.e., incoming and outgoing bandwidth and latency). 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, i.e., due to interconnections with hardware.

2.2 ThingsJS Framework

The ThingsJS Framework includes infrastructure components that are responsible for managing the dynamic execution of a ThingsJS

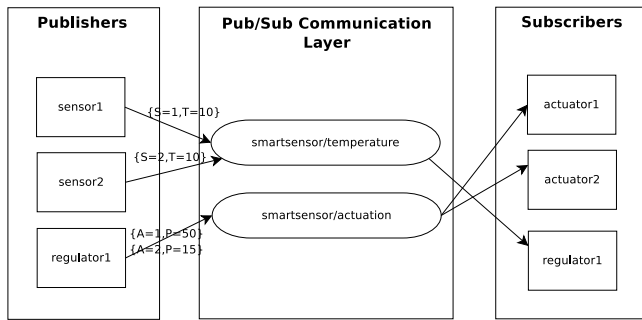


Figure 2: Topic-Based Publish/Subscribe Model

application over the set of devices that the IoT system comprises. The description of infrastructure components of the framework are described in the following subsections, and are shown in figure 1.

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 distributed execution across all participating ThingsJS devices. Note that the scheduling is done through the *ThingsJS Scheduler*, which is described in more details at 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 (more at section 3).

2.3 Inter-Component Communications

In *ThingsJS*, we require that all inter-component communications follow a topic-based publish/subscribe [8] (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 lightweightsness 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 [9], and also in a peer-to-peer fashion [5, 20]. 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, in order to optimize the communication flow depending on the dynamic placement of components to devices.

```

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

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 over topic *smartsensor/actuation*, which are then consumed by the actuator components to regulate the power output. Figure 3 illustrate sample pub/sub-related code for the *SmartTemperature* demo app, based on the modeling 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*) must come 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 at 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 while respecting the current execution of all other components on D_i . In future work, we plan on supporting other optimization criteria; i.e., minimizing component-to-component response time, bandwidth usage, costs, etc., as different components might have different optimization goals.

3.2 ThingsJS Scheduler: Challenges

Selecting the best device to host a given component is challenging, notably 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_i 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 proper 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 come up with 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, we use Machine Learning (ML) techniques to predict the execution time of a given component over all available devices $D_i \in D$. Then, we make use of a SMT SAT solver to determine the best assignment 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 [13, 19, 21, 21, 22], 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 MST SAT solver (i.e., [7]) to find the most suitable *global* configuration in an efficient manner. Note that since the scheduling occurs at the macro level, then 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 and must ensure that constraints are respected. Then, as the system is running and conditions change (i.e., constraints might be violated, or new component instances might be launched, etc.), then there might be a need to readjust the

scheduling by migrating already-executing components to different devices (see section 4). In that case, the *Scheduler* must also take into consideration the *costs* of migrating components to determine if the benefits will offset the costs. The design of a cost model will come in future work.

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.

An alternative approach would 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. In order to truly support dynamic and transparent code migration, there is a compelling need to come up with 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. Several of such techniques have been proposed in the literature [15, 18, 23, 24], 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, then process migration techniques might become impossible due to low-level platform differences (i.e., processor, memory-space, etc.).

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

```

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

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 capture and 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 fact that functions can 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*, proper to each *instance* of the closure.

From a graph theory perspective, the hierarchy of closure scopes form a tree-like structure [12] 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, [12] 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 its 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. Note that our prototype implementation is not fully done at the time of submission, and we are still working on some of the ideas. As we cannot easily access, serialize and deserialize the Javascript call stack nor the event queue, we *schedule* the saving of the state to be executed at the end of the event queue by means of timers. We think 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 proper 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.

5 VALIDATION PLAN

As this paper provides a holistic overview of what ThingsJS proposes and the various challenges that it solves, we do not yet provide experimental validation at this stage. We plan on deepening our research into the various areas outlined in this paper. Also, after completing our ThingsJS implementation, we plan on releasing it as an open-source project.

Prediction Model and Scheduler. We will thoroughly evaluate the prediction model and the SAT-based scheduler described at section 3. Our implementation will be built on top of the core *ThingsJS* implementation and will be deployed on several IoT devices (different models of Raspberry PIs). The evaluations will be carried under different device load profiles, and for different components. In addition to experimental validation over physical devices, we will also conduct simulated evaluations to assess the large-scale scalability of our model.

Javascript Migration Engine. We are already well advanced into the building of our Javascript migration framework. Upon our model and implementation being finalized, we intend on asserting its correctness, both over targeted demos as well as over real Node.js applications. We will also extensively measure the performance and overheads of our approach, compared with prior work, and across several IoT hardware profiles.

6 RELATED WORK

There has been some prior attempts at addressing the various challenges that *ThingsJS* aims at solving.

IoT Frameworks. In [10] the authors propose 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 [3, 17]. [16] surveys the main IoT middleware platforms and approaches.

Scheduling IoT Applications. To the best of our knowledge, *ThingsJS* is a first approach at proposing an IoT-specific prediction model and component scheduler. However, there has been some prior work in scheduling applications in a cloud setting [4, 11], but we think that using such approaches directly in our IoT context could be impractical, as there are typically far more IoT nodes in an IoT system compared to cloud systems, and that such nodes exhibit much more heterogeneity. As an example, the Cherrypick model [4] bears a complexity of $O(N^4)$, which is too high even for small IoT systems with only a few tenths of devices. The *ThingsJS Scheduler* is designed to support large-scale IoT applications across a wide range of heterogeneous devices.

Javascript Code Migration. [14] proposes a framework to migrate the state of browser-based Javascript applications across web browsers. [12] notably builds upon the prior approach by improving the handling of nested closures by instrumenting the VM to access its internal state. To the best of our knowledge, we found no approach that dealt with the specific problem of migrating Node.js out-of-browser applications. In addition, our approach attempts to avoid low-level VM instrumentation in order to increase cross-device and cross-VM portability.

7 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 aim at providing a rich platform that can abstract developers from scheduling, communications and fault tolerance considerations, at predicting the load of IoT applications and dynamically scheduling their placement across devices, and at enabling the transparent migration of rich IoT Javascript applications across heterogeneous IoT devices.

As future work, in addition to finalizing our first implementation and running experiments according to our validation plan, we would like to explore several other research directions. From a modeling perspective, we would like to look at how models from other paradigms (i.e., actor-based) could be integrated into ThingsJS, and how they could map to the pub/sub communication model. Another area would be to look at how we could provide an adaptive pub/sub model that could dynamically adapt to the communication flows

of all IoT components, considering their placement across devices. Finally, as many IoT systems are subject to security vulnerabilities [6], we would like to look at how we could integrate dynamic security features, such as live system-wide invariant monitoring.

REFERENCES

- [1] 2013. Redis Website. (2013). <http://www.redis.io/>
- [2] 2014. Intel XDK. (2014). <https://software.intel.com/en-us/xdk>
- [3] 2017. Node-RED Website. (2017). <https://nodered.org/>
- [4] 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.
- [5] 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. <https://doi.org/10.1109/JSAC.2002.803069>
- [6] 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.
- [7] 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>
- [8] 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>
- [9] 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>
- [10] Evgeny Gavrin, Sung-Jae Lee, Ruben Ayrapietyan, and Andrey Shitov. 2015. Ultra Lightweight JavaScript Engine for Internet of Things. In *SPLASH Companion 2015*. ACM, New York, NY, USA, 19–20.
- [11] 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.
- [12] 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.
- [13] Andy Liaw, Matthew Wiener, et al. 2002. Classification and regression by randomForest. *R news* 2, 3 (2002), 18–22.
- [14] James Teng Kin Lo, Eric Wohlstadt, 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.
- [15] Dejan S. Milošević, Fred Douglass, 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>
- [16] Anne H Ngu, Mario Gutierrez, Vangelis Metsis, Surya Nepal, and Quan Z Sheng. 2017. IoT middleware: A survey on issues and enabling technologies. *IEEE Internet of Things Journal* 4, 1 (2017), 1–20.
- [17] Per Persson and Ola Angelmark. 2015. Calvin—merging cloud and iot. *Procedia Computer Science* 52 (2015), 210–217.
- [18] 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>
- [19] Peter J Rousseeuw and Annick M Leroy. 2005. *Robust regression and outlier detection*. Vol. 589. John Wiley & sons.
- [20] 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.
- [21] Donald F Specht. 1991. A general regression neural network. *IEEE transactions on neural networks* 2, 6 (1991), 568–576.
- [22] 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.
- [23] Amirreza Zarrabi. 2012. A generic process migration algorithm. *International Journal of Distributed and Parallel Systems* 3, 5 (2012), 29.
- [24] 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>