ThingsMigrate: Platform-Independent Migration of Stateful JavaScript IoT Applications

Julien Gascon-Samson
Electrical and Computer Engineering Department, University of British Columbia, 2332 Main Mall, Vancouver, BC, Canada, V6T 1Z4
julien.gascon-samson@ece.ubc.ca

Kumseok Jung
Electrical and Computer Engineering Department, University of British Columbia, 2332 Main Mall, Vancouver, BC, Canada, V6T 1Z4
kumseok@ece.ubc.ca

Shivanshu Goyal
Electrical and Computer Engineering Department, University of British Columbia, 2332 Main Mall, Vancouver, BC, Canada, V6T 1Z4
shivanshu3@gmail.com

Armin Rezaiean-Asel
Electrical and Computer Engineering Department, University of British Columbia, 2332 Main Mall, Vancouver, BC, Canada, V6T 1Z4
armin.rezaiean.asel@gmail.com

Karthik Pattabiraman
Electrical and Computer Engineering Department, University of British Columbia, 2332 Main Mall, 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. As IoT devices become increasingly powerful, they can run more and more complex applications written in higher-level languages, such as JavaScript. However, by their nature, IoT devices are subject to resource constraints, which require applications to be dynamically migrated between devices (and the cloud). Further, IoT applications are also becoming more stateful, and hence we need to save their state during migration transparently to the programmer.

In this paper, we present ThingsMigrate, a middleware providing VM-independent migration of stateful JavaScript applications across IoT devices. ThingsMigrate captures and reconstructs the internal JavaScript program state by instrumenting application code before run time, without modifying the underlying Virtual Machine (VM), thus providing platform and VM-independence. We evaluated ThingsMigrate against standard benchmarks, and over two IoT platforms and a cloud-like environment. We show that it can successfully migrate even highly CPU-intensive applications, with acceptable overheads (about 30%), and supports multiple migrations.

2012 ACM Subject Classification Computer systems organization → Distributed architectures, Computer systems organization → Embedded and cyber-physical systems, Computer systems organization → Dependable and fault-tolerant systems and networks, Software and its engineering → Middleware, Software and its engineering → Process management, Software and its engineering → Functional languages, Software and its engineering → Language features, Software and its engineering → Publish-subscribe / event-based architectures

Keywords and phrases JavaScript, Code Migration, Closures, IoT, Node.js

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.18

32nd European Conference on Object-Oriented Programming (ECOOP 2018).
Editor: Todd Millstein; Article No. 18; pp. 18:1–18:32

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Funding This work is supported by a research gift from Intel, a Discovery grant and Post-Doctoral Fellowship from the Natural Sciences and Engineering Research Council of Canada (NSERC), and funding from the Institute for Computing, Information and Cognitive Systems (ICICS) at the University of British Columbia (UBC).

1 Introduction

The Internet of Things (IoT) involves multiple devices across many domains that are interconnected to provide and exchange data. Over the last few years, the IoT market has grown considerably with some estimates putting the number of IoT devices in the tens of billions [28]. IoT devices are becoming more and more powerful, and in many cases they can run full-fledged real-time operating systems (e.g., the popular Raspberry Pi can run full Linux distributions). As a result, future IoT devices will be able to execute stateful, distributed applications written in high-level languages, which provide greater abstraction and portability than those written using platform-specific APIs.

In this paper, we focus on the use of JavaScript for programming IoT devices. While JavaScript has enjoyed wide popularity in the web context for a long time, it is now a mature and rich language in its own right. It has also become more and more prevalent in the world of IoT due to its portability across a wide range of devices [27, 10], as well as its large installed base of libraries and developers who know the language. Further, the language's asynchronous nature makes it a prime candidate for IoT applications, which are often event-driven, and hence need to be highly reactive.

At the same time, placing more complex applications and long-running tasks on the end-devices themselves, close to the physical data (i.e., edge computing [38]) can incur lower latencies as opposed to running such applications in the cloud. However, as IoT devices are more resource-constrained, IoT applications have to be migrated between devices, and between devices and the cloud, for performance and security reasons. Thus, there is a compelling need to enable automated migration of stateful JavaScript devices between different IoT devices, and to/from the cloud, without requiring programmers to use platform specific APIs or runtimes. This is the main focus of this paper.

Migrating JavaScript applications poses several challenges, due to certain features of the language (i.e., closures) and its event-based nature. Further, due to the heterogeneity of IoT, we need to come up with a solution that does not involve accessing the internal states of the JavaScript Virtual Machine (VM), thus allowing portability across different devices. We tackle these challenges by proposing ThingsMigrate, a comprehensive middleware for the dynamic migration of IoT-based JavaScript applications across heterogeneous devices. ThingsMigrate automatically instruments the code at runtime to avoid modification of the VM while supporting advanced features of JavaScript, serializes its state, and reconstructs it on the target device after migration without any intervention from the programmer.

Other work has attempted to migrate browser-based JavaScript-based applications; however, they either do not fully address some important JavaScript features, such as nested closures ([32]), or they rely on VM-instrumentation [29] – thereby making their approach dependent on a specific VM/browser implementation. *To the best of our knowledge, ThingsMigrate is the first comprehensive high-level framework for migrating stateful JavaScript-based IoT applications, which addresses the aforementioned challenges, and without requiring any modifications to the JavaScript VM, thereby allowing platform-independent migrations.*

In summary, this paper provides the following contributions:

- A comprehensive JavaScript migration approach (Section 4) that is based on high-level

code instrumentation and reconstruction, and that does not require VM modification, thereby allowing cross-platform migrations of JavaScript-based IoT applications.
- System implementation (Section 5) that handles many advanced features of the language and environment, such as arbitrarily nested closures, event queues, timers and MQTT-based communication interfaces, and support for multiple migrations.
- Evaluation through the execution of benchmarks across IoT and cloud-based devices (Section 6). Results indicate that ThingsMigrate can instrument arbitrary JavaScript programs, serialize their state and reconstruct them in a reasonable amount of time, while incurring an execution time penalty of 30%. Further, ThingsMigrate was capable of supporting multiple migrations with minimal memory usage increase.
- A case study (Section 7) which describes the experience of applying our approach in a real-world IoT context (motion detection over a video stream), predominantly using third-party libraries developed for server applications.

2 Motivation

Use of JavaScript. We assume that the various software components to be executed on the IoT nodes are written in JavaScript. JavaScript is one of the most popular languages today (in 2018), and ranks sixth in the TIOBE programming languages index [13]. It has also been ranked as the top language on both *GitHub* and *Stack Overflow* for the last five years. While the predominant use of JavaScript is for the web, JavaScript also maps well to the asynchronous, event-based nature of IoT applications [39], which in turn simplifies the development of asynchronous and concurrent applications. Further, similar to other high-level languages, JavaScript runs over a VM and is platform-independent, which allows for run-time code portability. In fact, the use of JavaScript also opens the possibility of easily sharing code, data and development resources between the different components of the IoT and web software stacks (e.g., the client-side and server-side portions of end-user web applications in a Web of Things (WoT) setting could both be written in JavaScript) [26, 21, 31]. Further, as many IoT devices nowadays provide a browser-based interface, it is fair to assume that they will integrate a JavaScript VM.

IoT Devices are Resource-Constrained. As mentioned, there have been many attempts at either adapting existing JavaScript VMs (e.g., Node.js [42] for IoT devices), or developing new JavaScript VMs [27, 10, 3, 12] for the IoT. However, given the resource-constrained nature of IoT devices and the fluid nature of the resource constraints, applications running on such devices might have to be frequently migrated from one device to another. For example, when a device runs low in memory, then the application running on it should be migrated to another device with more available memory to avoid the application from running out of memory and crashing. Similarly, any change to the available bandwidth or to the computational load of a given IoT device might require network and delay-sensitive applications to be migrated to a different device. Migration may also be needed when there are external factors causing device failures (e.g., device gets overheated or physically damaged), or due to security attacks on IoT devices.

Further, while resource management and code migration is a well understood problem in classical and cloud-based distributed systems, we believe that these techniques are not directly applicable to the IoT landscape, as they do not take into account IoT-specific constraints such as the wide heterogeneity of hardware and software platforms, the highly resource-limited nature of the devices which makes it impractical to introduce additional virtualization layers, and the limited ability to provision resources on demand [41]. In addition, as the compute

capacity of IoT devices is dictated by energy efficiency [30], we believe that a great deal of flexibility is required in scheduling. Therefore, given these considerations, we believe a static deployment of IoT applications to devices is insufficient, and that there is a need for applications to be migratable.

Preserving the State. As IoT devices and applications become more complex, they inherently generate more elements of *state* (i.e., variables, arrays, objects) as part of their execution. For instance, for an application which detects motion patterns in a video stream (Section 7), elements of state would include the pixels of the currently processed video frames (arrays), as well as any intermediate results produced as part of the computation. Considering that migration may be triggered at any time during the application's execution in response to changing resource conditions or external events such as failures, it is important that there be a transparent mechanism to serialize and deserialize the state of an executing application. This mechanism should be efficient and support general JavaScript applications for IoT with only minimal modifications. Otherwise, developers would be required to implement application-specific serialization and deserialization logic, and for any arbitrary point in the execution, which would complicate the application logic.

Migration support in the VM. While migration support could ultimately be implemented in the VM, we believe that this is unlikely in the near future given the vast heterogeneity in IoT platform ecosystems. Unlike in the web browser space where there are only a handful of dominant players, the JavaScript IoT landscape is much more fragmented, with the availability of a wide variety of JavaScript engines (e.g., [27, 10, 3, 12]). Further, migration support would be required at both ends of the migration process; i.e., at the source device/VM, and at the target device/VM, which may be different from each other. Some of the VMs may be closed-source and hence not easily modifiable. That being said, should full or partial support for migration be provided in the VM or as part of the ECMAScript standard (i.e., by enabling special APIs to access the state of closures), then our technique could be adapted accordingly.

Applicability to other Languages. In this paper, we focus on the migration of JavaScript code for IoT. While our solution is specifically tailored for the challenges raised in migrating JavaScript applications (closures, objects, timers, events, etc.), we believe that some of the techniques that we propose could be adapted to support the migration of code written in other high-level languages. For instance, Python also provides support for closures and hence, we believe that our closure serialization and reconstruction approach could be adapted to Python. However, we do not consider applications written using low-level languages such as C or assembly. We also assume that developers do not use low-level APIs to directly access hardware state of IoT devices (e.g., by reading the pins of a device in a platform-specific manner) as such code would be difficult to migrate.

3 System Model

The system architecture of ThingsMigrate is presented in Figure 1. It is derived from the architecture of ThingsJS, a comprehensive IoT middleware that we presented as a vision paper in [26][1] (more details are given in appendix A). Our system model assumes a set of IoT devices, which are in charge of executing the various components of a distributed IoT application. We describe the systems components in this section.

[1] While ThingsJS proposes migration as part of an integrated system, it does not specifically address migration challenges.

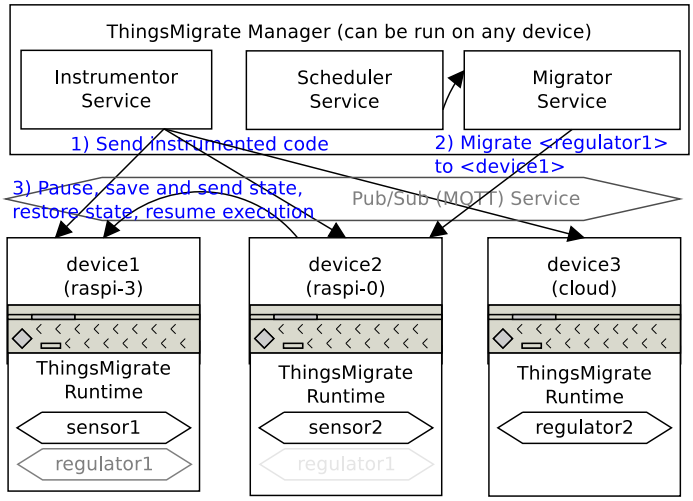


Figure 1 High-Level Architecture of ThingsMigrate

3.1 ThingsMigrate Manager

The central piece of our architecture, namely the *ThingsMigrate Manager*, manages the execution of distributed IoT applications across the set of available devices. In our model, all communications between the components of the system use the topic-based publish-subscribe (pub/sub) paradigm (also referred to as MQTT) [24], which enjoys widespread usage in the IoT world [25, 40]. This is because it allows decoupling content producers (*publishers*) from content consumers (*subscribers*), and allows for abstracting network considerations. Overall, the *Manager* component has three components:

(1) Scheduler. This component schedules the execution of all IoT components across all devices. For the Scheduler to operate efficiently, developers are encouraged to modularize their IoT applications into a set of components, and to follow the best practices of JavaScript (Section 4.1). Taking into consideration the capabilities of each device, the requirements of the components, and a set of developer-specified *constraints*, the scheduler assigns the execution of each component onto a specific device. Upon the conditions changing, the scheduler can decide to dynamically move some of the components between devices. The migration takes place dynamically, and preserves the state of JavaScript IoT applications, so that the execution can be transparently transferred from one device to another - this is our main contribution. Note that the details of the *Scheduler* are outside the scope of this paper.

(2) Instrumentor. This component is in charge of instrumenting the JavaScript source code of the IoT components, which is the code that is executed by the devices. This is executed at the beginning before running a component on ThingsMigrate.

(3) Migrator. The *Migrator* is in charge of transparently migrating the execution of each component from the original to the target device. To migrate a given component (e.g., component *regulator1* on device 2), the migrator issues a migrate command to the *ThingsMigrate Runtime* running on the target device (Section 3.2), with the name of the component to migrate. This then triggers the migration.

3.2 ThingsMigrate Runtime

The *ThingsMigrate Runtime* is a thin JavaScript middleware that executes on each IoT device and manages the local execution of all the components running on the device. It

```
1   function CreateCounters(n) {
2      var total = 0;
3      function Counter() {
4         var value = 0;
5
6         return function() {
7            value += 1;
8            total += 1;
9
10           // Can access parent variables
11           console.log("val=" + val + " value=" + value + " total=" + total);
12           return value;
13        }
14     };
15
16     var counters = [];
17     for (var i=0; i<n; i++)
18        counters.push( Counter() );
19     return counters;
20  }
21
22  var counters = CreateCounters(2);
23  setInterval(function() {counters[0]},1000);
24  setInterval(function() {counters[1]},500);
```

Figure 2 Counters JavaScript Example

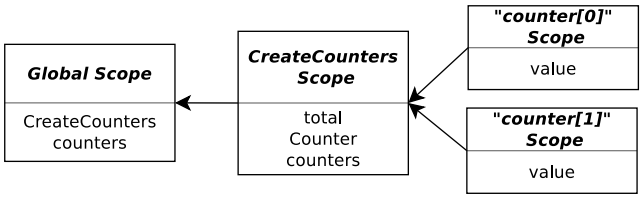


Figure 3 Counters JavaScript Example Closures and Scopes

receives and executes the instrumented source code of the various components that it needs to execute from the *Instrumentor*, and awaits migration commands from the *Migrator* over a pub/sub interface. Upon a *migrate* command being received for a component (e.g., for component *regulator1* on device 2), the Runtime first freezes the execution of the component, serializes its state (Section 4.6) and sends it to the target device over the pub/sub interface (e.g., *device1*). When the Runtime on the target device (e.g., *device1*) receives the serialized state for a component, it restores the serialized state by generating appropriate restoration code (Section 4.7), which allows the execution to be resumed with the pre-migration state.

4 Approach

As mentioned, our code migration approach works entirely at the JavaScript layer through code instrumentation and does not depend on the underlying VM (unlike prior work by Kwon et. al. [29]). Thus, to support code migration, we need to instrument the code to expose the internal states (i.e., closures) that are not directly accessible using the JavaScript primitives and reflection APIs (Section 4.5). Code migration works in three phases. First, the component's code is instrumented to support migration. Second, upon the *Migrator* service triggering a migration, a *snapshot* of the current state is taken (Section 4.6) and transmitted to the target device. Finally, the target device reconstructs the component state based on the snapshot, and resumes execution (Section 4.7).

4.1 Assumptions

We assume that the JavaScript code is compliant with strict-mode ES5 (ECMAScript 5) [5], which has been the de facto standard for many years. Although ES6 ([4]) is gaining momentum, it mostly adds syntactic sugar over ES5. Support for ES6 can easily be provided by leveraging transpilers (e.g., Babel.js [1]) to convert to ES5.

Because JavaScript is event-driven and single-threaded, we assume that developers will avoid blocking the main thread for long periods of time, as this would prevent the migration from being scheduled. Note that this assumption is not specific to ThingsMigrate – in fact, a long-running operation that never yields the control would inhibit the dispatching of any asynchronous event (e.g., timers, messages, I/O). To use ThingsMigrate, developers should follow the best practices and write their code in an event-driven manner, or break long-running operations to yield control (e.g., `setImmediate`) at periodic intervals, so that event processing can take place.

Finally, we assume that the program uses publish-subscribe (pub/sub) for communication, and does not perform write operations to the local file system. The former assumption is common in the IoT world, while the latter assumption requires the programmer to send file system operations over the network (Section 4.8 has more details).

4.2 Motivating Example

The JavaScript language treats functions as data, and provides support for closures, which allows for functions to be defined in other functions and to be bound to variables. As such, like any other object, functions can be passed as parameters and can be *returned* within other functions. JavaScript closures can also access the variables defined in parent functions in addition to their own variables, even if a parent function goes out of scope.

Figure 2 presents a motivating example of using JavaScript closures to implement a simple counter `Counter` (lines 3-14) which returns a *function* (lines 6-13) that increments the counter's value by 1 (line 7) and prints it (line 11). In addition, there is a global variable `total` that holds the sum of all counters (line 8). Further, we wrap the `Counter` function and the `total` variable in another function, `CreateCounters`, which allows for creating and returning an array of n counters. More precisely, n nested closures are returned, which upon being called, increment the corresponding counter; therefore, the variable `counters` holds an array of $n = 2$ counters (line 20). Note that after line 20, some variables become out of scope (e.g., `total`, and all the copies of `value` for each counter), but they are not garbage-collected as the nested counter incrementation functions (i.e., `counters`) still access them.

Figure 3 visually illustrates the *scopes* of the various closures and their relationship. As can be observed, there are two independent copies of variable `value` each defined in their own scope, but only one copy of `total`, which is defined in the parent scope and is hence *shared* with the two child scopes. Finally, there are two recurrent timers (set using the `setInterval` JavaScript function) which increment the two counters at a regular interval i.e., by invoking nested functions `counters[0]` and `counter[1]` respectively every 1000 ms and 500 ms.

4.3 Challenges

Migrating the execution of a JavaScript program from one VM to another VM on to a different device poses many challenges when it comes to capturing and reconstructing the state. We discuss the challenges below in the context of the motivating example. To support migration, the current state of the application must be serialized. A naive approach to serialization would be to dump the process space of the application, which comprises the heap

and the stack. However, such an approach would require serializing the entire memory space of the process, which would be platform-dependent and inefficient. Rather, ThingsMigrate provides a JavaScript-based approach that exploits the specifics of the language. For instance, as mentioned, a JavaScript application is made of objects and closures. More specifically, there is one root object that contains a set of properties, which are in fact objects themselves. As JavaScript treats functions as data, it allows functions to be bound to and stored within objects (i.e., *closures*). However, as shown in the motivating example above, functions can also contain other objects stored in variables.

(1) Closures. While JavaScript includes APIs to recursively and dynamically walk through the properties of objects and serialize them, the state of closures is *hidden* and thus cannot be *accessed* by means of user code. Thus, we need mechanisms to dynamically expose these hidden parts of the state *during* program execution.

(2) Migrating Events. We also need mechanisms to seamlessly transfer the state of pub/sub interfaces (i.e., subscribers and publications) during a migration. In addition, IoT systems often perform delayed executions (i.e., using timers); therefore, we need to support the seamless migration of timer-based events.

(3) Handling the Call Context. As JavaScript is mostly single-threaded and asynchronous (i.e., event-driven), there is no easy way to interrupt the current execution to perform a migration. In addition, as the call stack is not exposed, it is not directly accessible. Therefore, we need to come up with a mechanism to trigger the migration at certain points in the execution of the program.

(4) Reconstructing the state. After migrating the state, the execution must be restored given the serialized state. This is non-trivial, as an equivalent reconstructed program must be generated from the original code *and* the serialized state, and the execution must resume exactly at that state without any side effects (i.e., without re-executing code that can potentially lead to a different outcome).

(5) Enabling Multiple Migrations. As a given program might be migrated multiple times, we need to support multiple migrations. To reach that goal, the reconstructed program must be generated in such a way that it can be migrated again, with low overheads. For example, the reconstructed program should not incur significantly higher memory or performance overheads than the original program, as such overheads would quickly add up when performing multiple migrations.

4.4 Problem Statement

As mentioned, in order to capture the state of a JavaScript program, one needs to capture the hierarchy of *scopes*, starting from the global scope, as well the data elements (variables and functions) contained within each scope. In other words, ThingsMigrate captures the *structure* and the *values* of the different state elements. More formally, we denote the state of a JavaScript application as $\mathbb{S} = \langle S, F, V, R \rangle$, where S is the set of *scopes*, F is the set of *functions*, V is the set of *variables* (i.e., a tuple of $\langle \text{name}, \text{value} \rangle$) and R is the set of *relations* between scopes and other entities (i.e., a tuple of $\langle \text{scope}, \text{entity} \rangle$).

Taking the code snippet shown in Figure 2 as an example, and assuming that a snapshot of the state is taken after 3250ms, then two instances of the `Counter` function (and their associated scopes) are defined, due to the timer invocations (i.e., `Counter_1` and `Counter_2`). Also, there are two instances of the anonymous function defined inside `Counter` (i.e., `Counter_1_anon` and `Counter_2_anon`). However, there is only one copy of the `CreateCounters` function (i.e., `CreateCounters_1`). Further, there are two copies of variable `value`, each within its own scope (`Counter_1_value` and `Counter_2_value`), and

one copy of variable `total` (`CreateCounters_1_total`). The resulting state of the snapshot object $\mathbb{S} = \langle S, F, V, R \rangle$ would respectively contain states S, functions and their definition F (omitted for brevity), variables and their value V, and the set of relationships R between each variable/function and its associated scope:

$$
\begin{aligned}
S =& \{\texttt{global}, \texttt{CreateCounters\_1}, \texttt{Counter\_1}, \texttt{Counter\_2}\} \\
V =& \{(\texttt{global\_counters}[0], \texttt{Counter\_1\_anon}), (\texttt{global\_counters}[1], \texttt{Counter\_2\_anon})\}, \\
& \{(\texttt{CreateCounters\_1\_total}, 9), (\texttt{CreateCounters\_1\_counters}[0], \texttt{Counter\_1\_anon})\}, \\
& \{(\texttt{CreateCounters\_1\_counters}[1], \texttt{Counter\_2\_anon}), (\texttt{Counter\_1\_value}, 3)\}, \\
& \{(\texttt{Counter\_2\_value}, 6)\} \\
F =& \{(\texttt{global\_CreateCounters}, ...), (\texttt{CreateCounters\_1\_Counter}, ...), (\texttt{Counter\_1\_anon}, ...)\}, \\
& \{(\texttt{Counter\_2\_anon}, ...)\} \\
R =& \{(\texttt{global}, \texttt{CreateCounters\_1}), (\texttt{global}, \texttt{global\_counters}), (\texttt{global}, \texttt{global\_CreateCounters})\}, \\
& \{(\texttt{CreateCounters\_1}, \texttt{Counter\_1}), (\texttt{CreateCounters\_1}, \texttt{Counter\_2})\}, \\
& \{(\texttt{CreateCounters\_1}, \texttt{CreateCounters\_1\_total}), (\texttt{CreateCounters\_1}, \texttt{CreateCounters\_1\_counters})\}, \\
& \{(\texttt{CreateCounters\_1}, \texttt{CreateCounters\_1\_Counter}), (\texttt{Counter\_1}, \texttt{Counter\_1\_value})\}, \\
& \{(\texttt{Counter\_1}, \texttt{Counter\_1\_anon}), (\texttt{Counter\_2}, \texttt{Counter\_2\_value}), (\texttt{Counter\_2}, \texttt{Counter\_2\_anon})\}
\end{aligned}
$$

For more details, we refer the reader to Section 4.7 (Phase 3: Code Restoration), which describes in more detail our algorithmic approach to generating reconstruction code, and which gives an example of the restored code of the same code sample (Figure 2), migrated after the same delay (3250ms). As can be observed, the same functions, scopes and variables, as well as their relationships, are depicted in the restored code sample (Figure 5).

The next sections describe the algorithmic process followed by ThingsMigrate to (1) instrument the code to expose the hidden states, (2) take a snapshot and (3) reconstruct the code at the serialized state.

4.5 Phase 1: Code Instrumentation

In the code instrumentation phase, the ThingsMigrate Runtime augments the input JavaScript source file to allow the state to be dynamically captured (challenge 1), corresponding to the formal model defined in Section 4.4. Our code instrumentation approach is inspired by the work in Lo et. al. [32], but differs significantly as Lo et. al. [32] only offers limited support for capturing and restoring complex closures. In the example shown in Figure 2, Lo et. al. [32] would be able to capture and restore the scope of the two internal *counter* closures, but would not accurately model the relationship between said scopes in the restored output, so that two instances of the `CreateCounters` scope would be generated rather than one, ending with two distinct `total` variables after restoration. Each nested scope would then update its own `total` variable, which would be inaccurate.

The main aspects of our technique are illustrated in Algorithm 1. To fully capture the state of closures, the *Instrumentor Service* exposes the scope hierarchy by injecting code at relevant locations that will mirror the chaining of scopes and their contents (i.e., functions and variables) in a parallel tree-like data structure, in order to expose and capture the state.

Upon requesting the instrumentation of a given JavaScript source file (lines 1-8), an Abstract Syntax Tree (AST) representation of the code is first generated. The algorithm starts at the root node of the AST tree (line 3) and recursively iterates over the child nodes. Note that as a convention, throughout this algorithm, lines that start with the symbol < and end with > represent code that is *injected* in the form of AST nodes *at that particular location in the AST tree processing*, to augment the input code.

```
function instrumentCode (sourceCode)
begin
    rootNode ← ASTParse(sourceCode)
    <setupMigrationListener()>
    <globalScope ← Scope(rootNode.name, null)>
    instrumentNode(rootNode, null)
    return ASTGenerate(rootNode)
end
function instrumentNode (parentNode, parentScope)
begin
    foreach node in parentNode do
        if node is Function then
            <scope ← Scope(node.name, parentScope)>
            instrumentNode(node, scope)
            <parentScope.addFunction(node)>
            <scope.checkAndDestroy()>
        end
        else if node is VariableDeclaration then
            <parentScope.addVar(node.name, node.value)>
        end
        else if node is VariableAssignment then
            varScope ← findScope(parentScope, node.name)
            <varScope.setVar(node.name, node.value)>
        end
    end
end
```

Algorithm 1: Code Instrumentation

```
var global = new Scope("global");
function CreateCounters(n) {
    var createcounters = new Scope(global, "CreateCounters");
    var total = 0;
    createcounters.addVar("total", total);
    function Counter() {
        counter. = new Scope(createcounters, "Counter");
        var value = 0;
        counter.addVar("value", value);

        var anon1 = function() {
            anon1 = new Scope(createcounters, "anon1");
            value += 1;
            anon1.setVar("value", value);
            total += 1;
            anon1.setVar("total", value);

            // Can access parent local variables
            console.log("val=" + val + ", value=" + value + " total=" + total);

            return value;
        }
        counter.addFunction("anon1", anon1);
        return anon1;
    };
    createcounters.addFunction("Counter", Counter);

    var counters = [];
    CreateCounters.addVar("counters", counters);
    for (var i=0; i<n; i++) {
        counters.push( Counter() );
        CreateCounters.setVar("counters", counters);
    }
    return counters;
}
global.addFunction("CreateCounters", CreateCounters);

var counters = CreateCounters(2);
CreateCounters.setVar("counters", counters);
ThingsMigrate.setInterval(function() { counters[0] }, 1000);
ThingsMigrate.setInterval(function() { counters[1] }, 500);
```

Figure 4 Counters JavaScript Example - Instrumented

The general idea of the algorithm is that for each function, a `Scope` object is instantiated *in the output code*, and linked to the parent scope, so that an exposed scope tree can be built dynamically *at the time of execution*. Upon the algorithm starting, a `Scope` referring to the global scope is generated (line 5), without any parent scope. Processing then starts from that root node (first invocation of `instrumentNode` at line 9). Then, for each child node in the AST tree, if the child node is a *function*, we generate a new Scope linking back to the parent scope (line 13), and we recursively invoke `instrumentNode` again for that node. We also *register* the function in its parent scope, which will allow us to dynamically retrieve it at the serialization phase (Section 4.6), as otherwise, there would be no way to dynamically access it (as the JavaScript reflection API does not allow access to functions, variables and scopes).

For similar reasons, if the current node being parsed corresponds to the declaration of a new variable, the corresponding variable must be added to the node's current scope (line 19). For an operation that would set the contents of a variable (assignment, incrementation, etc.), we must also refresh the corresponding variable in the scope in which it was declared, to make sure that its content is mirrored in the tree (lines 22-23). Note that this operation first requires finding the scope in which the variable was declared, due to the JavaScript execution model in which a variable defined in any parent scope can be accessed by any child scope (e.g., variable `total` in Figure 2). To that end, the `findScope` function (line 22) walks the tree upwards until it encounters the *most recent* declaration of the variable (up to the global scope). The value of the variable is then updated in that scope.

Figure 4 shows a simplified instrumented version of the original source code shown in Figure 2 (note that in our implementation, many more details are included, which are omitted here for the sake of brevity). The lines of code that are added by the code instrumentation process are shown in grey. As can be observed, all defined scopes (the global scope, then the scopes corresponding to each function definition) are mirrored through an instance of a ThingsMigrate `Scope` object (lines 1, 3, 7 and 12). In addition, each variable definition or assignment gets mirrored in the tree, in the scope at which it is defined (lines 5, 9, 14, 16, 29, 32 and 39). Similarly, functions are also registered (lines 23, 26 and 36).

Instrumenting Timers. Following a similar algorithmic approach as in Lo et. al. [32], ThingsMigrate provides support for saving the state of timer functions, namely `setInterval` and `setTimeout` (challenge 2). This is accomplished in the instrumentation phase by replacing standard timer calls by invocations of our own functions, which expose the state of the timers at serialization time. Thus, at restoration time, the timers resume at the state when serialization took place. For instance, considering our code example, if snapshotting occurs after 250ms, then the first timer (line 23) will first trigger after 750ms, then every second, while the second timer (line 24) will first trigger after 250ms, then every 500ms.

Pub/Sub Interfaces. ThingsMigrate provides support for capturing the state of pub/sub interfaces (challenge 2). Similar to how we handle timers, ThingsMigrate wraps calls to the pub/sub interface (MQTT library) at the instrumentation phase, so that upon a migration being requested, the *list of each topic previously subscribed* by the application gets serialized as part of the snapshot. Then, at the restoration phase, prior to resuming the execution, a subscription is transparently reestablished to each of the previously subscribed topics. To ensure that no publications are lost *during the migration*, we assume that reliable pub/sub is provided by the service [44, 20], so that the latter can retransmit any missed publication sent during the migration.

In addition, as the migration is triggered by a pub/sub publication, the *Instrumentor Service* injects code in the header to setup a pub/sub listener for the migration, when the instrumented program is executed. Upon the specific publication arriving, the framework

starts the state serialization process.

Classes and Prototypes. JavaScript ES5 does not support classes *per se* unlike object-oriented languages (e.g., Java). Instead, it provides high-level abstractions that emulate classes by means of *prototypal inheritance* [23]. ThingsMigrate provides support for serializing JavaScript-like ES5 classes by serializing each object's *prototype object*, so that upon restoring the code, the correct prototypal chain can be recreated along with the objects.

Cleaning Orphaned Scopes. During the life cycle of a JavaScript application, scopes are dynamically created, and can sometimes become *orphaned*. Orphaned scopes are scopes for which there are no single remaining reference to them or to one of their child scopes. In the example shown in Figure 2, at each timer iteration (lines 23-24), the function scope that is created on the fly (first argument) becomes orphaned and is therefore destroyed, as its serialization will not be required. Therefore, we need to destroy the scope objects corresponding to orphaned scopes, as they can lead to memory size increase – this problem is exacerbated on multiple migrations (challenge 5).

As a novel contribution, ThingsMigrate provides support for automatically destroying orphaned scopes, to support multiple migrations (challenge 5) on the same application without increasing the snapshot size and incurring additional overhead in the restored code (i.e., scope explosion). In the instrumentation phase, prior to any given function ending or returning, an API call to `scope.checkAndDestroy()` (line 16 of Algorithm 1) is *injected*, for the current `scope` object. At execution time, this function will check whether any other scope or variable depend on this scope. If there are no dependencies, then the scope is destroyed, and therefore it will not be serialized in the snapshotting phase (Section 4.6).

4.6 Phase 2: Snapshotting and Migrating

To trigger a migration, the component that is being executed receives a *migrate* pub/sub command from the *Migrator Service*. Recall that the code instrumentation phase sets up a listener, which initiates the migration (Section 4.5).

Serializing the State. The migration process first involves serializing the state to the JSON format. To do so, the scope tree is recursively walked in a top-down approach, from the global scope. The serialized output includes, for each scope, the variables and parameters, as well as nested scopes and functions. In JavaScript, functions cannot be serialized as-is. Thus, upon encountering a function when walking the scope tree, the function is assigned a unique ID, and the function's source code is added to a table of functions, which is appended at the end of the serialized state. Note that the serialized output also contains the state for special objects that ThingsMigrate addresses, such as timers and pub/sub interfaces.

Handling the Stack. We address the challenge of handling the stack (challenge 3) by exploiting the asynchronous, event-driven nature of JavaScript. Because JavaScript applications are single-threaded and are event-based, the runtime maintains an event queue. We *schedule* code migrations as events so that they get pushed at the end of the event queue and get executed over an *empty* stack. More precisely, as migration requests are sent through the form of pub/sub publications, they are treated as events and pushed to the event queue. Note that we could also accomplish the same behavior by scheduling the migration as a timer-based event.

Sending the Serialized State. Once the snapshot is generated, it is sent over the pub/sub interface to the target IoT node, which will regenerate the code considering the state of the snapshot, and resume execution.

4.7 Phase 3: Code Restoration

Upon a given IoT node receiving a snapshot, it needs to reconstruct the original program *at the exact state* where migration took place (challenge 4). The code restoration process must retain the original program structure, while reassigning the values for constructs holding state, such as variables, parameters and closures, without directly restoring the memory regions - this is important for platform independence and portability.

Reconstructing Closures and Scopes. As in the code instrumentation phase, closures pose unique challenges when it comes to generating restoration code, as they wrap state elements. Because functions can be return values of functions in JavaScript (e.g., as seen in Figure 2), there can exist multiple *copies* of a function sharing the same code, but corresponding to different *states* (i.e., holding different values). The code restoration process needs to generate multiple copies of some of the function trees, as state can be held not only in the functions themselves, but anywhere in parent functions as well, and bind such copies; i.e., to variables or parameters. For instance, in Figure 3, two counters are defined (i.e., `counter[0]` and `counter[1]`), which both point to a function having the same source code (i.e., the anonymous function at lines to 13), but holding different states, as the value of `value` defined in the parent function (`Counter`) is different. Thus, in the reconstructed code, two copies of `Counter` and its inner function (i.e., the *chain of functions*) will need to be defined to expose the different scopes of the two counter closures.

```
1  function generateScope (scope, parentScope)
2  begin
3      < (function(){ >
4      <var {scope.name} ← Scope(scope.name, parentScope)>
5      foreach param in scope.params do
6          <var {param.name} ← param.value>
7      end
8      foreach function in scope.functions do
9          <functionTables[scope].code>
10     end
11     foreach variable in scope.variables do
12         if scopeDefinitionExists(variable.value) then
13             <var {variable.name} ← variable.value>
14         end
15         else
16             stage2Variables.add(variable)
17         end
18     end
19     foreach child in scope.children do
20         generateCode(child, scope);
21     end
22     foreach variable in stage2Variables do
23         <var {variable.name} ← variable.value>
24     end
25     < }()) >
26 end
```

Algorithm 2: Code Generation

Code Generation Algorithm. A simplified version of the code generation algorithm is shown in Algorithm 2. In an nutshell, the algorithm starts with the global scope (function), and recursively reconstructs the scopes in a hierarchical manner. For a given scope, it first injects the parameters defined in that scope with their values at snapshot time (lines 5-7), then injects the full source code for the functions defined in that scope, *including the function headers* (lines 8-10). Then, the variables defined or redefined in that scope are injected and set to their value at snapshot time (lines 11-18). In some corner cases involving JavaScript

```
1  /* Original code comes before */
2  function() {
3     function CreateCounters(n) {
4        var n = 2;
5        function Counter_1() {
6           var anon1 = function() { /* ... */ }
7           ThingsMigrate.addFunction("Global/CreateCounters/Counter_1/anon1",
                anon1);
8           var value = 3;
9           return anon1;
10       }();
11
12       function Counter_2() {
13          var anon1 = function() { /* ... */ }
14          ThingsMigrate.addFunction("Global/CreateCounters/Counter_2/anon1",
                anon1);
15          var value = 6;
16          return anon1;
17       }();
18
19       var total = 9;
20       var counters = [Counter_1, Counter_2];
21    }(2);
22 }();
23
24 ThingsMigrate.setInterval(ThingsMigrate.findFunction("Global/CreateCounters/
       Counter_1/anon1", 1000, 250);
25 ThingsMigrate.setInterval(ThingsMigrate.findFunction("Global/CreateCounters/
       Counter_2/anon1", 500, 250);
```

Figure 5 Counters JavaScript Example - Restored Code

objects and their prototypes, it might happen that some scopes cannot be resolved for some of the variables, at the first (i.e., top-down) phase. An example would be a case where an object instance is constructed, but the constructor function is defined in a child scope that is not yet generated (i.e., invoked) at the time of assigning the variable. Our approach addresses these situations by placing such variables in a queue, to process them in a later stage (i.e., at stage 2, after the generation of the child scopes - lines 22-24). After generating the variables, the `generateScope` function is recursively called for all child scopes of the current scope (lines 19-21).

Each scope definition is *wrapped* in an enclosed `(function() {...} ())` call, which means the scope definition code (i.e., the output of the algorithm) will be *invoked* when the restored code is executed (lines 3, 25). In other words, the nested scope generation portions of code will be invoked recursively, thereby recreating the scope hierarchy. Note that the *functions* themselves corresponding to each scope are not executed upon restoration, as this could lead to side effects (i.e., non-determinism). It is nevertheless necessary to include their definitions, as they might be invoked later in the code *after restoration*.

Code Restoration Example. Assume that a snapshot was taken after executing the code shown in Figure 2 for 3.25 seconds. Figure 5 illustrates the restored code. Note that while this example has been derived from the output of a real invocation of the code restoration procedure of ThingsMigrate, some simplifications and adjustments were made for clarity. Also, the names of the various entities within this snippet (i.e., variables, functions, scopes), as well as their relationships, correspond to the state example shown in Section 4.4.

As can be observed, two copies of the `Counter` closures have been generated: `Counter_1` and `Counter_2` (lines 5-10 and 12-17), which both wrap the values for variable `value`: 3 and 6 (as the timers triggering the closure incrementation functions were invoked respectively 3 and 6 times - the former every second, and the latter every 500 ms). Similarly, there is only one instance of the `CreateCounters` closure, which is the parent of the two `Counter` functions (line 3). It holds the `total` variable (`total = 3 + 6`, line 19). Upon executing the

restored code, the `CreateCounters`, `Counter_1` and `Counter_2` functions are re-executed, thereby recreating the closures as before.

Note that upon restoring a function, we add it to a function table, which will allow us to refer to it later (not shown in Algorithm 2). As an example, at lines 7 and 14, we add the restored closures that correspond to the anonymous counter incrementation functions to the function table, and we then retrieve them from the table when we restore the timers (i.e., lines 24-25), as the timers periodically invoke these functions.

Multiple Migrations. ThingsMigrate supports transparent multiple migrations without introducing additional overhead (challenge 5). This is accomplished at the code restoration phase by maintaining a unique scope tree structure that is accessed by all the generated closures and scopes, and by re-injecting scope definitions (i.e., variables, parameters, nested functions, etc.) across the regenerated code, following an approach derived from Algorithm 1. Further, relevant pub/sub code is re-injected to support receiving *migrate* messages again. In other words, the output of the code restoration phase is *an alternate code segment* equivalent to the output of the code instrumentation phase, which can hence support further migrations.

4.8 Limitations

Handling External Libraries. ThingsMigrate does not yet provide full support for imported libraries (i.e., the `require` statement). A simple solution would be to directly import the code in the main JavaScript module itself prior to instrumentation. This approach may be inefficient however, if there are multiple levels of nested library imports. Another solution would be for ThingsMigrate to provide a migration interface, and for module developers to implement the interface for either a more optimized migration of the nested libraries, or for supporting libraries exposing native I/O resources, such as file system access. Despite this limitation, we find that ThingsMigrate can support many third-party libraries as we show in Section 7.

Scope Explosion. If programs make use of several levels of nested closures, then the resulting snapshot and restored code can become quite large, due to the phenomenon of *scope explosion*, in which multiple scopes might have to be maintained. However, this problem is symptomatic of bad programming practices and is not specific to ThingsMigrate, as the JavaScript VM itself will have to retain a large amount of scope structures in-memory.

Redirecting I/O Operations. As mentioned in Section 4.1, ThingsMigrate assumes that all communications are done over the pub/sub interface. Further, in the current state, ThingsMigrate does not support file I/O operations, which is non-trivial, as reads and writes must occur where the corresponding files are located. For instance, assume there is a file on device A which is read by an application on the same device that gets migrated to device B. In order to guarantee consistent reads, one has to migrate not only the current position in the file, which is trivial, but also to guarantee (1) the availability of the file on B, or (2) to provide some redirection mechanism.

As JavaScript I/O operations are typically handled through streams, we plan on transparently redirecting streams over the pub/sub interface (solution 2 above), by wrapping the base JavaScript stream API (similar to wrapping timer-based or pub/sub-based APIs). A stream-level solution can support arbitrary stream-based I/O operations, such as files, network, and even HTTP requests. Upon device A receiving a migration request to migrate a given app to device B, the ThingsMigrate Runtime will generate a unique ID for each currently active stream, and will setup a transparent forwarding mechanism over a pub/sub bridge (i.e., by creating a topic corresponding to that ID that both devices A and B will subscribe to). Then, upon a read operation being requested by the app on device B, for

a given stream, the request will be transparently forwarded by the Runtime to device A, who will perform the read and send back the results to the Runtime on B, who will deliver them to the stream at the application layer. Likewise, any write operation will simply be forwarded from the Runtime on B to the Runtime on A, who will complete the write.

Nested Timers. A limitation of ThingsMigrate occurs in the handling of some deeply nested timer-related calls (i.e., `setTimeout`, `setImmediate`). Should a *snapshot* command be received while a *timer* is in a *pending* state – i.e., before the callback function is invoked – then the timer gets cleared, the remaining time and the reference to the callback function are serialized, and migration happens normally. However, should the snapshot command be received *after* the callback function is invoked, then a race condition occurs between any asynchronous calls made inside the body of the callback and the snapshot function. Race conditions are sometimes problematic in JavaScript, as the ordering of events can't always be predicted [16, 33]. For instance, should the JavaScript VM event loop process the snapshot function before the asynchronous calls, then the resulting snapshot will not contain the scopes created by the asynchronous calls, producing an incorrect snapshot. Handling nested timers would require that the snapshot function be delayed until all callbacks have been resolved, which is a non-trivial problem. As a potential solution, we propose to inject, at the instrumentation phase, specific code into the function scope that will signal the function's completion, which would allow us to detect the resolution of nested asynchronous calls.

5 Implementation

ThingsMigrate is implemented in the form of a JavaScript library[2] that can be included by the application. Its implementation is built over ThingsJS [26] (more details in appendix A). It provides APIs that can be invoked to perform code instrumentation, snapshotting and code restoration. From a higher-level perspective, it also provides an execution environment that replicates the architecture shown in Figure 1. More specifically, it provides a Runtime environment that can be run on IoT devices supporting an appropriate VM (e.g., Node.js on Raspberry PIs Models 3 and 0), as well as a *Manager* component, which is used to transparently instrument JavaScript programs, launch them on specific IoT nodes (decided by a scheduler), monitor them, and trigger a serialization/migration. Internally, our implementation uses the popular `esprima` library [7] to parse JavaScript code into an AST, and the `escodegen` [6] to convert back an AST into JavaScript code.

We also provide a web dashboard to monitor the execution of the application on different devices, and trigger the migration at runtime (more details in appendix B).

6 Experimental Validation

We perform three experiments to validate ThingsMigrate. Experiment 1 (Section 6.2) benchmarks the performance of our code instrumentation algorithm against a set of benchmarks. Experiment 2 (Section 6.3) measures the performance overhead of ThingsMigrate for benchmarks running on different devices. Finally, Experiment 3 (Section 6.4) evaluates the multi-migration capabilities of ThingsMigrate by migrating a benchmark application several times, across several devices.

[2] `http://www.github.com/karthikp-ubc/thingsjs`

6.1 Experimental Setup

ThingsMigrate provides JavaScript migration between IoT devices, and between devices and the cloud. To emulate different scenarios, we ran our experiments on two IoT platforms, namely a Raspberry Pi model 3B (quad-core 1.2 Ghz ARM7, 1 GB memory), and a Raspberry Pi model 0W (single-core 1 Ghz ARM6, 512 MB memory), both running the Raspbian Jessie operating system (a Debian Linux variant). We also included a cloud server (Xeon E3-1220 v3, quad-core 3.10Ghz, 32 GB memory). All nodes were running the Node.js VM version 6, which is ES-5 compliant. While we did not test other VMs due to stability issues or to their lack of compliance with the ES-5 standard, the Node.js VMs we used were all compiled differently for each target platform.

Despite an extensive search, we did not find publicly-available sets of IoT-specific JavaScript benchmarks to evaluate our system. Prior work ([39]) has built their own IoT-specific JavaScript benchmarks[3]. We followed a similar approach and built two IoT-specific benchmarks: (1) a *factorial* application, which computes the factorial of a very large number and uses closures to store the computed digits (i.e., in a very large expanding array), and (2), a *regulator* application, which models an IoT *edge* component which receives temperature measurement data from different sensors[4] over a pub/sub interface, keeps the previous n values for m sensors, and periodically computes an optimal power adjustment to be sent to an actuator. `factorial` models a CPU and memory-intensive application of a finite duration (experiment 2), while `regulator` models a less intensive (i.e., low CPU and memory usage) application that runs for a long time. We note that the memory usage of the `regulator` is similar to the memory usage of the IoT-specific benchmarks described in [39].

In addition, for experiments 1 and 2, we also used some benchmarks from the Chromium Octane [2] suite, which were originally designed to stress-test the performance of the V8 JavaScript engine in the Chrome web browser. While they are not representative of IoT applications, we nevertheless use them to assess the universality of our framework, and for performance testing of ThingsMigrate under extreme conditions.

6.2 Experiment 1: Code Instrumentation

In this experiment, we consider all the benchmark programs from the Chromium Octane suite that do not depend on a web browser (i.e., accessing the DOM or any other in-browser object), as ThingsMigrate migrates IoT applications rather than in-browser applications. We measure the time it takes to instrument the code for these benchmarks[5], as well as for our `factorial` and `regulator` applications. In addition, we compare the size of the uninstrumented (raw) code, and the size of the instrumented code. Results are shown in Table 1. As one can observe, the instrumentation algorithm executes quickly (in under 1 second), even for the complex benchmark applications with large code sizes. Further, code instrumentation is a one-time process for any given program.

The increases in the code size due to instrumentation range from 26.9% to 7382.7%, with an average of 1174.1%. This is because the instrumentor assigns human-readable variable and function names in the generated code for debugging purposes - this can be reduced by using a minifier [14]. We do not deploy these techniques. However, the code size has minimal impact on the runtime performance as JavaScript is compiled just-in-time.

[3] The source code is not publicly available, and hence we cannot use them.
[4] We fed the application with random values, as the computed result itself is not part of the experiment.
[5] Measurements were taken on our cloud server.

Benchmark	Program Size (kb)	Instrumented Size (kb)	Instr. Time (ms)
navier-stokes	9.985	122.263	135.67 ± 4.5
splay	6.573	45.984	86.18 ± 2.7
deltablue	1.5452	115.623	120.65 ± 0.6
crypto	39.028	276.763	194.05 ± 1.6
box2d	357.169	2773.027	821.95 ± 3.0
earley-boyer	159.794	574.463	301.9 ± 0.8
raytrace	24.998	31.720	64.2 ± 0.5
richards	8.302	59.922	87.4 ± 0.5
typescript	2.138	10.541	31.75 ± 0.2
factorial	0.952	5.526	28.21 ± 0.2
regulator	1.855	15.594	42.1 ± 0.4

Table 1 Code Instrumentation Results (with a confidence interval of 95%)

6.3 Experiment 2: Performance Overhead

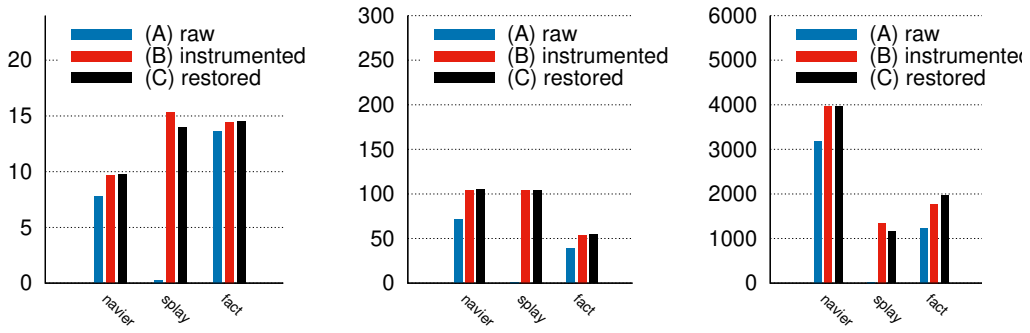
In this experiment, we analyze the performance impact of ThingsMigrate over a set of highly resource-intensive benchmarks. The goal of this experiment is to model the execution of a resource-intensive task of a finite duration (i.e, eventually returns a result) that would be executed over different IoT devices and the cloud server. We selected benchmarks `navier-stokes` and `splay` from the Octane suite, as they respectively model extreme conditions, in terms of CPU usage and memory utilization. Further, we were successful in running these benchmarks on all test devices, unlike most other benchmarks in the suite (even without our instrumentation, most of the benchmarks in the Octane suite were unable to run on the Rapsberry Pi 0 due to its limited capabilities). We also used our `factorial` application.

For each benchmark program, we measure and compare the time to complete its execution. For each benchmark, and for our 3 target devices (Raspberry Pi 3, Pi 0 and our cloud server), we run (A) the non-instrumented (raw) code, (B) the instrumented code, and (C) the code generated after migration[6]. Further, we measure the average memory usage of each application for the same three versions of the benchmark (raw, instrumented and generated) to determine memory overheads. Finally, we report the time taken to serialize and generate restoration code. We ran each benchmark until 50% of its execution time, took a snapshot, generated restoration code from the snapshot, and then resumed the execution with the restored code. Each benchmark was executed multiple times on each platform, and the times averaged over the executions were reported.

Execution Time. Execution time results are shown in Figure 6. For both `navier-stokes` and `factorial`, we observe an execution time overhead ranging from 5% and up to 40% compared with the raw code (A), for all devices, which is due to the overhead of our injected instrumentation code. This is because these applications have a significant amount of state.

As for the `splay` benchmark, the performance of the instrumented (B) and the restored code (C) was significantly degraded. This is due to the extreme amount of memory operations that the benchmark performs, which significantly slows down the execution. This slowdown

[6] As results for the restored code (C) were only available after completing a migration (i.e., at a given time t during execution), results for (A) and (B) were considered also only after time t in their respective runs, for a fair comparison.



(a) Cloud Server (Xeon) **(b)** Raspberry Pi 3 **(c)** Raspberry Pi 0

Figure 6 Execution Time (in seconds). Margins of errors were below 1.5% for most of our results, and up to 6% for some of our results on the Pi 0, for a confidence interval of 95%.

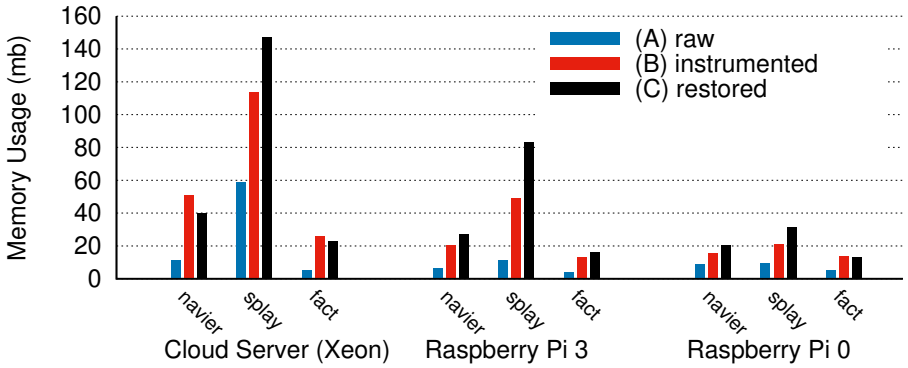


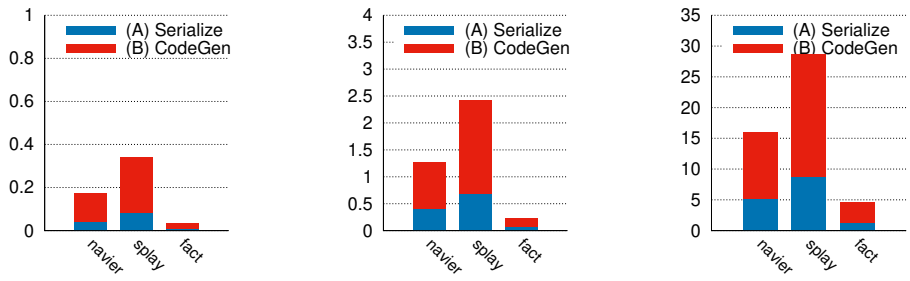
Figure 7 Memory Usage (mb). Margins of errors are not shown, as the results show the averaged memory usage for all runs, averaged over the duration of the experiment.

is amplified by the mirroring of the scope tree, which consumes even more memory. Further, as our *Pi* devices have much slower memory, compared to our cloud server, the performance overhead is higher. We stress however that these benchmarks were specifically designed to model *extreme* conditions on desktop computers, and are not typical applications to be run on IoT end nodes, which are much more resource constrained.

We also observe that the performance of the instrumented code (B) and the restored code (C) is roughly similar across all benchmarks. As the restored code is *semantically equivalent* to the original code, but with instrumentation to enable further migrations, we obtain similar performance as the instrumented *non-migrated* code. These results indicate that the performance (i.e., execution time) *will not* degrade after migration (Section 6.4).

Unfortunately, we cannot perform direct comparisons with prior work in terms of execution time overhead, as Lo et. al. [32] measured such overheads for web applications on desktop computers, which do not exhibit the same workload characteristics as our benchmarks, and Kwon et. al. [29] did not report the execution time overheads of their programs at all.

Memory Usage. Our memory overhead results are depicted in Figure 7. For each benchmark and device, we averaged the memory usage over time across the duration of each execution, and we report averaged results for all experimental runs. Overall, our results reveal that executing the instrumented code (B) significantly increases the memory usage



(a) Cloud Server (Xeon) **(b)** Raspberry Pi 3 **(c)** Raspberry Pi 0

Figure 8 Serialization / Code Generation Time (in seconds). Margins of errors were between 0.5% and 5% for all results, for a confidence interval of 95%.

compared to the non-instrumented code (up to 6 times). This is expected, as we do not rely on JavaScript VM instrumentation at runtime (unlike Kwon et. al. [29]), therefore many more elements of state must be captured during execution and mirrored. In addition to maintaining the scope hierarchy that mirrors the closures, the instrumented code also maintains a copy of every variable, parameter and function, which increases the memory usage. The results for `factorial` exhibit a similar trend across all devices, with the restored code (C) having a slightly lower memory footprint compared to the instrumented code (B). After restoration (C), we start with a fresh *instrumented* copy of the code (at snapshot time), without the *past* states that potentially contain portions that were not yet garbage collected.

On the other end, `navier-stokes` and `splay` exhibit much higher memory usage for the restored code (C) compared to the instrumented code (B). As these benchmarks are more aggressive in stressing the memory (i.e., by allocating and deallocating scopes), the resulting reconstruction code is very verbose (i.e., due to the explicit definition/duplication of nested closures as shown in Figure 5). Therefore, despite being *semantically* equivalent as the instrumented code at snapshot time, the reconstructed code may be harder to optimize by the VM. The execution of the JavaScript Garbage Collector (GC) can also be a cause of the overhead. Experiment 6.4 discusses the effects of the GC on the live execution of JavaScript applications. We stress again that such benchmarks represent extreme, non-typical conditions. Nevertheless, they could run even on low-end devices (e.g., the Raspberry Pi 0), and the average memory footprint remained under 30 MB.

We also note that the memory usage for both Raspberry Pi devices are much lower than the cloud server. This is attributable to the *bitness* of the devices; i.e., our cloud server has a 64 bit processor, while the other Pi devices are 32 bits, and a more aggressive GC execution on the Pi devices, as they are more memory-constrained.

Finally, prior work (i.e., [32, 29]) did not evaluate the runtime memory overhead of their approach and hence, we cannot compare our results against them.

Serialization and Code Reconstruction Time. Considering the same experimental setup (i.e., same devices and same benchmarks as above), we measured the time it took to serialize the state at mid-experiment, and to generate restoration code from the state. Results are shown in Figure 8. The results exhibit the same trend across all platforms, and vary based on the size and complexity of the application. Overall, the serialization and snapshot times are reasonable, albeit slightly higher on the Raspberry Pi 0 and for the two Octane benchmarks. We stress that the Raspberry Pi 3 device is roughly 7 times slower than our cloud device, while the Pi 0 device is roughly 90 times slower than our cloud server.

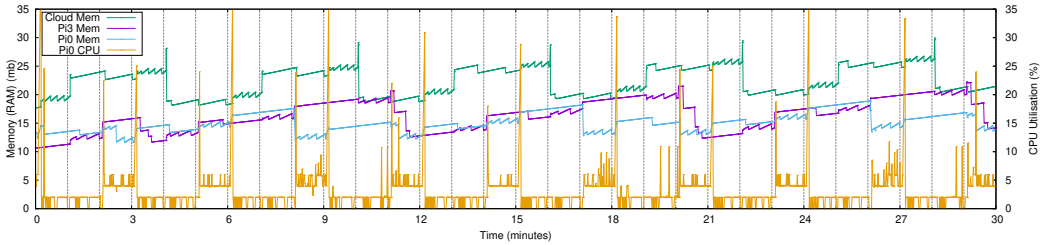


Figure 9 Multi-Hop Migration Analysis (*regulator* application)

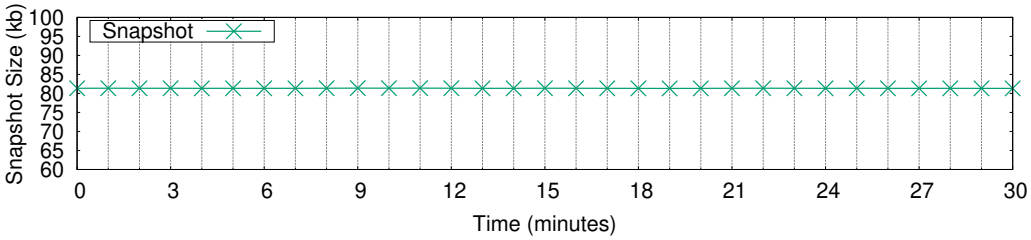


Figure 10 Snapshot Size over Time (*regulator* application)

6.4 Experiment 3: Multiple Migrations

In this experiment, we analyze the global behavior and performance over time of Things-Migrate, when multiple migrations are performed between the *edge* and the *cloud*. More precisely, we analyze the effects of migrating a long-running task that is not computationally expensive from one device to another. None of the benchmarks used in Experiment 2 fit this description, nor could we find publicly available JavaScript-based IoT benchmarks that satisfy this criteria (Section 6.1). Therefore, we developed and used our own benchmark – the *regulator* application that satisfies this criteria (by design). We first deploy the regulator application on our cloud server, then we migrate it to the *edge* devices (i.e., the Raspberry Pi 3 device, after one minute, and then to the Pi 0 device, after one minute). The application is then pushed back to the cloud server. This cycle is repeated 10 times (30 migrations over 30 minutes), and the CPU and memory utilization are measured in each instance.

The memory utilization results are shown in Figure 9, for the duration of the experiment (30 min). The migration cycles are denoted by a vertical bar (every minute), and an oscillating variation pattern can be observed during the time periods for which each device was executing the regulator application. As can be observed, the memory usage fluctuates, for all devices, but remains overall stable, as each successive code restoration does not consume additional memory (assuming the memory needs of the application do not increase). The step-like appearance of the memory curves are explained by the JavaScript garbage collector (GC), which regularly claims small amounts of memory (i.e., during execution of the *regulator* – small pikes), and which periodically runs a more through collection (bigger drops). However, we also observe that the memory tends to very slowly increase over time, but this is not due to the multiple migrations – rather, this is an artifact of the experimental data collection process, which logs memory and CPU usage at a frequent interval (every 200ms) and keeps the data in memory. This is supported by Figure 10, which plots the snapshot size at each successive migration, which remains constant at 83kb. Finally, as in Experiment 2 (Section 6.3), the memory usage on the cloud server is higher than on the pi devices.

The CPU usage is shown on the same Figure (9). For simplicity, we show CPU usage

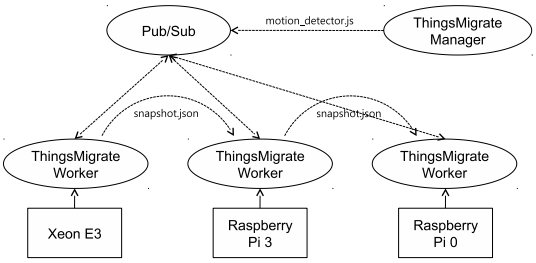


Figure 11 Case study setup

results only for one device (i.e., Pi 0, which is the most resource constrained), but the trend is similar on the others. As can be observed, the CPU usage peaks at about 4%-5% when the Pi 0 device is executing the application, and is close to 0% otherwise. The CPU usage during execution remains constant across the different executions. The short spike *before* execution corresponds to the code reconstruction, and the short spike *after* execution corresponds to the serialization process, for which a small memory surge can also be observed.

6.5 Summary

Overall, our results demonstrate that ThingsMigrate can enable the cross-platform migration of IoT JavaScript-based applications with acceptable performance overhead (~30% for normal cases), and without any modifications to the underlying VM. While the memory overheads are more significant, we believe that this is an acceptable tradeoff given the goal of our approach to rely purely on code instrumentation. We also believe that memory gains could be achieved by optimization techniques such as storing references to variables rather than copying them within the scope tree. However, this is a subject for future exploration. Further, our results show that ThingsMigrate was able to handle multiple-hops migrations while keeping the CPU and memory usage almost constant.

7 Case Study: Motion Detector

In this section, we describe our experience with using ThingsMigrate to build a realistic IoT application for video surveillance by adapting third-party JavaScript components developed for standalone node.js applications. These components were not designed with ThingsMigrate in mind, and as a result, we had to make (minor) modifications to make them work with our system. We also evaluate this application using application-specific metrics that are more likely to be of interest to end users rather than CPU/memory usage (unlike Section 6).

7.1 Experimental Setup

We set up an IoT network with four devices to build a surveillance system. Figure 11 shows the setup. The application logic is modularized into two components: a video streamer component that captures images from a video source such as a webcam, and a motion detector component that processes the images to detect motion. Unlike the video streamer, which is bound to a single device by the peripheral from which it needs to capture video, the motion detector can be run on any device as it performs computations on the image data. We measured the behavior of the system over a series of migrations of the motion detector across the three systems (Raspberry Pi 3, Pi 0, and the cloud server from Section 6).

Video-streamer: We used `FFmpeg` [8], a popular open-source software for handling multimedia, to capture individual frames from a video stream. For the purpose of the experiment, the component was configured to stream from a video file instead of a peripheral such as a webcam, so that we have a deterministic and reproducible sequence of frames. To interface with the `FFmpeg` process from the JavaScript layer, we adapted a third-party NPM library called `fluent-ffmpeg` [9], that we use to capture individual frames and publish them over the pub/sub interface. The capture-and-publish routine was written as a single JavaScript function `captureFrame` that was passed into a `setInterval` call with interval set to 200ms (i.e., a rate of 5 frames per second). We used the cloud server to serve as a surveillance camera and run the video streamer component.

Motion Detector: This component was written entirely in JavaScript without having to interface with any external software. We integrated a third-party NPM module called `jimp` [11], which provides an API to read `Buffer` objects (i.e., received from the pub/sub overlay) and perform image processing tasks. The component stores binary frame data for the n latest frames.

The motion detection logic (i.e., function `detectMotion`) iterates through the array of images and computes the difference between subsequent frames by calling `jimp.diff()`. The binary difference between the frames is published over the pub/sub interface. In addition, if more than 10% of the pixels are altered, a motion *detected message* is also published. The `detectMotion` function is passed to a `setInterval` call with the interval set to 500ms - this is lower than the frame rate of the video streamer (Section 7.2 explains why). Since the `detectMotion` works by retrospective inspection of past frames, the array of `Buffer` objects containing the image data needs to be migrated. Otherwise, the restored component would need to wait for the buffer of past frames to be filled again – thereby skipping the motion detection process for a given time window, and missing potentially important motion.

Although we do not fully support the migration of external libraries (Section 4.8), it was possible to integrate the third-party NPM libraries as the objects they created were native JavaScript objects and the API calls were limited to stateless operations. For instance, since the Buffer objects are native objects, they could be easily serialized and migrated. The function call to `jimp.diff()` is a stateless operation, since it does not create any additional scopes and its execution context is destroyed after it returns. Such stateless operations do not affect the migration process because we do not need to serialize their scopes.

Finally, we collected performance statistics by subscribing to a pub/sub topic at which each ThingsMigrate runtime publishes its CPU and memory usage. To monitor and verify that the motion detection was working correctly, we used our web dashboard, which displays the images by converting the data into a base64 encoded PNG image.

7.2 Results

To automate our migration test in a controlled fashion, we wrote a Node.js script to send commands to the IoT devices over the pub/sub interface. We sent a *migrate* command every 1 minute to the Cloud Server, Pi 3, and Pi 0, and back. We repeated the cycle 3 times.

Figure 12 shows the CPU usage over time as the application is migrated between the devices. The collected data for CPU and memory usage across the three devices exhibit a similar pattern to the regulator component discussed in Section 6.4. The CPU usage on a device has a spike upon receiving a snapshot and just before sending a snapshot, remains high while it is running a component, and stays near 0 during the idle state. The memory consumption stays within a narrow range, with the garbage collector being triggered more frequently while a device is running a component, and occasionally while it is idle.

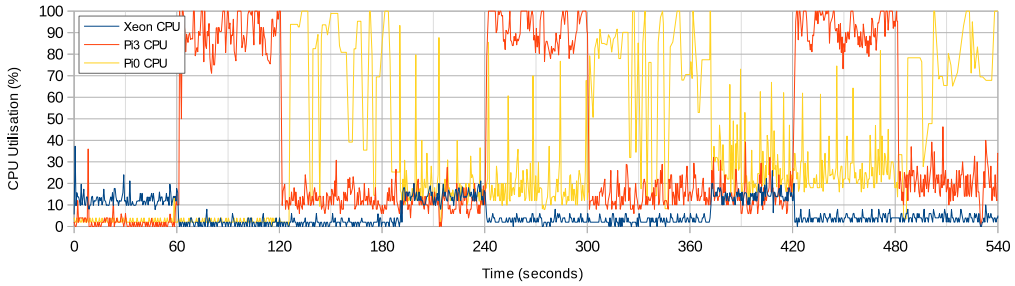


Figure 12 CPU Usage over time – Motion Detector component

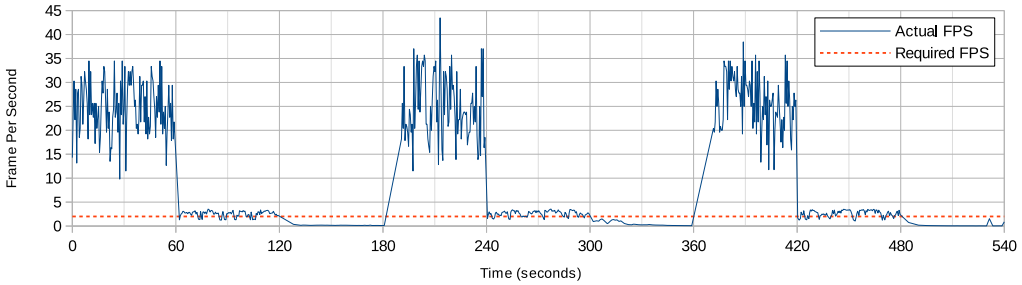


Figure 13 FPS over time – Motion Detector component

However, on the Raspberry Pi 0's console, we observed error messages showing that the process failed at regular intervals. This is because the asynchronous call to *detectMotion* took much longer than the set interval of 500ms, due to the limited computational capacity of the Pi0, which led to the event queue accumulating faster than the JavaScript VM could consume, which eventually led to overflowing and halting of the program.

Figure 13 shows the frame rate measured in Frames Per Second (FPS) on each device over time. The FPS was calculated using the formula $\frac{1}{\Delta t}$ where Δt is the time taken to execute the *detectMotion* function. The figure shows the FPS dropping below the required FPS of 2 over the periods between 120 and 180, 300 and 360, and 480 and 540 seconds, during which the Pi 0 was running the motion detector component. We can also observe the *detectMotion* function blocking the thread at 180 seconds and 360 seconds, preventing the migration from being triggered. The FPS drops below 2 occasionally during Pi 3's execution, but for most frames it is able to process within the time interval, compensating for the delay overall.

In summary, this case study shows that we were able to successfully migrate a third-party application with minimal modifications between different devices. We also found the frame rate measured in FPS was acceptable in most cases for the application. However, special considerations might have to be taken for low-end devices, as migration requests can be delayed due to delays in running more intensive tasks. These issues should be considered when designing a system-level solution as discussed in Section 3.1.

8 Related Work

There has been some prior work in the area of migrating the execution of JavaScript code. However, they focus on migrating web applications between web browsers [18, 32, 36, 29], and hence have very different constraints from IoT devices. Imagen [32] migrates web applications

across heterogeneous browsers without altering the VM, and address some of the challenges specific to web applications (e.g., the DOM, HTML5 media elements, timers). However, their handling of nested closures is limited (Section 4.5). In [29], extending their prior work in [36], the authors provide deeper support for serializing and reconstructing closures for migrating web applications, but they require VM instrumentation to access the internal scope tree, which makes their approach less portable as it is bound to a specific browser version of an open-source Webkit browser. In contrast, our approach does not require any modifications to the JavaScript VM, and is hence platform neutral. Further, we provide explicit support for multiple migrations, which prior work does not.

As an alternative to capturing and restoring the state of the web application, deterministic replay techniques can be used to replay an exact sequence of actions leading to the current state [17, 34, 37, 19, 15]. However, these approaches focus on capturing and replaying web browser events, and are not directly applicable to IoT environments. Further, they may even be impractical for IoT environments which have limited resources, as the sequence of events to be captured and replayed often grows rapidly over time [32].

There have been many attempts at providing low-level code migration techniques that directly save and restore the process memory space, and are hence programming language independent [35, 45, 46]. Such techniques could be applied for migrating JavaScript code, but they would require serializing the state of the JavaScript virtual machine (VM) itself, which can incur significant overheads on IoT devices. Further, considering as per our model that one VM might host several components, this would make it difficult to separate the per-component state. Finally, providing platform independent migration would not be possible, as even the same version of a given VM might have different cross-platform implementations and memory layouts due to hardware differences. Note that similar challenges can be found in migrating virtualized OSes across devices [22, 43].

9 Conclusion and Future Work

In this paper, we presented ThingsMigrate, a middleware layer that provides VM-independent migration of stateful JavaScript applications across IoT devices. ThingsMigrate uses code instrumentation to expose the hidden states of a JavaScript application, thereby allowing its state to be captured and serialized, without requiring VM instrumentation. ThingsMigrate then generates a reconstructed version of the same application *at the serialized state*, allowing its execution to continue on a different device. We built an implementation of ThingsMigrate and evaluated it on three different devices, and against both standard benchmarks and custom applications. Our results show that ThingsMigrate can instrument, serialize and reconstruct JavaScript applications within reasonable time bounds, depending on the state and complexity of the input application. Further, we find that ThingsMigrate imposes an average 30% execution time overlay at runtime, which is reasonable given the non-reliance on VM-dependant low-level techniques (i.e., VM instrumentation). Finally, we show that ThingsMigrate supports multiple migrations across different devices without incurring additional overheads.

As future work, we intend on improving support for more complex cases of classes and prototypes, as well as supporting the features of the newer ECMA standards. We would also like to accomplish migration without interrupting the execution flow (i.e., seamless migration). Another interesting area would be to adapt our approach to provide fault tolerance in an IoT setting. While this could be provided by periodically saving the state to a reliable entity, we would like to explore the problem of *dynamically* serializing the state to persistent storage *during* execution.

References

1 Babel.js javascript compiler, 2017. URL: https://babeljs.io/.

2 Chromium octane benchmark suite, 2017. URL: https://github.com/chromium/octane.

3 *DukTape*. 2017. URL: http://www.duktape.org/.

4 *ECMAScript 2015 Language Specification*. 2017. URL: http://www.ecma-international.org/ecma-262/6.0/.

5 *ECMAScript 5.1 Language Specification*. 2017. URL: http://www.ecma-international.org/ecma-262/5.1/.

6 *escodegen: ECMAScript code generator*. 2017. URL: https://github.com/estools/escodegen.

7 *esprima: ECMAScript parsing infrastructure for multipurpose analysis*. 2017. URL: http://esprima.org/.

8 Ffmpeg website, 2017. URL: https://www.ffmpeg.org.

9 Fluent ffmpeg-api for node.js, 2017. URL: https://github.com/fluent-ffmpeg/node-fluent-ffmpeg.

10 Intel xdk, 2017. URL: https://software.intel.com/en-us/xdk.

11 Jimp: Javascript image manipulation program, 2017. URL: https://github.com/oliver-moran/jimp.

12 *mjs*. 2017. URL: https://github.com/cesanta/mjs.

13 Tiobe index. https://www.tiobe.com/tiobe-index/, 2017.

14 node-minify - npm, 2018. URL: https://www.npmjs.com/package/node-minify.

15 Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. Understanding javascript event-based interactions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 367–377. ACM, 2014.

16 Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for javascript. *ACM Computing Surveys (CSUR)*, 50(5):66, 2017.

17 Silviu Andrica and George Candea. Warr: A tool for high-fidelity web application record and replay. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 403–410. IEEE, 2011.

18 Federico Bellucci, Giuseppe Ghiani, Fabio Paternò, and Carmen Santoro. Engineering javascript state persistence of web applications migrating across multiple devices. In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, pages 105–110. ACM, 2011.

19 Brian Burg, Richard Bailey, Andrew J Ko, and Michael D Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 473–484. ACM, 2013.

20 Tiancheng Chang and Hein Meling. Byzantine fault-tolerant publish/subscribe: A cloud computing infrastructure. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 454–456. IEEE, 2012.

21 Ioannis K Chaniotis, Kyriakos-Ioannis D Kyriakou, and Nikolaos D Tselikas. Is node. js a viable option for building modern web applications? a performance evaluation study. *Computing*, 97(10):1023–1044, 2015.

22 Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

23 Douglas Crockford. *JavaScript: The Good Parts: The Good Parts*. " O'Reilly Media, Inc.", 2008.

24 Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003. doi:10.1145/857076.857078.

25 Paul Fremantle, Benjamin Aziz, Jacek Kopeckỳ, and Philip Scott. Federated identity and access management for the internet of things. In *Secure Internet of Things (SIoT), 2014 International Workshop on*, pages 10–17. IEEE, 2014.

26 Julien Gascon-Samson, Mohammad Rafiuzzaman, and Karthik Pattabiraman. Thingsjs: Towards a flexible and self-adaptable middleware for dynamic and heterogeneous iot environments. In *Proceedings of the 4th Workshop on Middleware and Applications for the Internet of Things*, M4IoT '17, pages 11–16, 2017.

27 Evgeny Gavrin, Sung-Jae Lee, Ruben Ayrapetyan, and Andrey Shitov. Ultra lightweight javascript engine for internet of things. In *SPLASH Companion 2015*, pages 19–20, New York, NY, USA, 2015. ACM.

28 Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

29 Jin-woo Kwon and Soo-Mook Moon. Web application migration with closure reconstruction. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 133–142, Geneva, Switzerland, 2017.

30 Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 234–251. ACM, 2017.

31 Jimmy Lin and Kareem El Gebaly. The future of big data is... javascript? *IEEE Internet Computing*, 20(5):82–88, 2016.

32 James Teng Kin Lo, Eric Wohlstadter, and Ali Mesbah. Imagen: Runtime migration of browser sessions for javascript web applications. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 815–826, New York, NY, USA, 2013. ACM.

33 Magnus Madsen, Ondřej Lhoták, and Frank Tip. A model for reasoning about javascript promises. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):86, 2017.

34 James W Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for javascript applications. In *NSDI*, volume 10, pages 159–174, 2010.

35 Dejan S. Milojíčić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32(3):241–299, September 2000. URL: http://doi.acm.org/10.1145/367701.367728, doi:10.1145/367701.367728.

36 JinSeok Oh, Jin-woo Kwon, Hyukwoo Park, and Soo-Mook Moon. Migration of web applications with seamless execution. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, pages 173–185, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2731186.2731197, doi:10.1145/2731186.2731197.

37 Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498. ACM, 2013.

38 Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

39 Dongig Sin and Dongkun Shin. Performance and resource analysis on the javascript runtime for iot devices. In *International Conference on Computational Science and Its Applications*, pages 602–609. Springer, 2016.

40 Meena Singh, MA Rajan, VL Shivraj, and P Balamuralidhar. Secure mqtt for internet of things (iot). In *Communication Systems and Network Technologies (CSNT), 2015 Fifth International Conference on*, pages 746–751. IEEE, 2015.

41 A. Taivalsaari and T. Mikkonen. A roadmap to the programmable world: Software challenges in the iot era. *IEEE Software*, 34(1):72–80, Jan 2017. doi:10.1109/MS.2017.26.

42 Stefan Tilkov and Steve Vinoski. Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010.

43 Timothy Wood, Prashant J Shenoy, Arun Venkataramani, Mazin S Yousif, et al. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, volume 7, pages 17–17, 2007.

44 Young Yoon, Vinod Muthusamy, and Hans-Arno Jacobsen. Foundations for highly available content-based publish/subscribe overlays. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 800–811. IEEE, 2011.

45 Amirreza Zarrabi. A generic process migration algorithm. *International Journal of Distributed and Parallel Systems*, 3(5):29, 2012.

46 S. Zhongyuan, Q. Jianzhong, L. Shukuan, and Z. Qiang. 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)*, pages 50–53, Aug 2015. doi:10.1109/DCABES.2015.20.

A ThingsJS and ThingsMigrate

As mentioned previously, the implementation of our system (ThingsMigrate) was realized over ThingsJS [26], our general-purpose framework for executing high-level *edge* applications on IoT devices. In this appendix, we provide a brief summary of ThingsJS below, and its relationship with ThingsMigrate. We then give some details on our open-source implementation of ThingsMigrate/ThingsJS.

A.1 Architecture

The high-level architecture of the ThingsJS Framework consists of several distributed components and is presented in Figure 14. From a holistic point of view, a ThingsJS environment comprises a highly-distributed ThingsJS *Application*, and dynamically manages its execution over a set of heterogeneous *devices* through the ThingsJS *Framework*. More details on ThingsJS are available in our vision paper [26].

ThingsJS Manager. The ThingsJS Manager is the center-piece of our system architecture, and manages the execution of all components across all devices. It takes as input a ThingsJS application, written in JavaScript, and schedules and monitors its distributed execution across all participating ThingsJS devices. The Manager can decide to trigger the migration of a given application towards another device – to accomplish this, it uses the ThingsMigrate APIs.

ThingsJS Runtime. An instance of the ThingsJS Runtime is present on every device and acts as a thin hypervisor layer. It locally manages the execution of all components on the device, and gathers detailed statistics during execution (CPU, memory and bandwidth usage) which are fed to the *Manager*. Upon the migration of a given application being requested, ThingsMigrate coordinates the migration through the *Runtime* components on both devices involved.

Inter-Component Communications. ThingsJS provides a pub/sub-based communication substrate (MQTT), and requires that all inter-component communications follow that model. The choice of this model was primarily motivated by the logical decoupling of content producers from content consumers that it provides. Like any other component, ThingsMigrate makes use of the pub/sub communication substrate for all communications (i.e., migration commands and snapshots are transferred directly between relevant Runtime nodes through the pub/sub interface).

ThingsMigrate. As a subcomponent of ThingsJS, ThingsMigrate provides support for dynamically migrating JavaScript IoT applications between devices, and is the focus of this paper. More details on the architecture on ThingsMigrate and its components are given in Section 3 and Figure 1 of this paper. In our specific implementation, as scheduling considerations are outside the scope of our paper, we let the user deploy applications manually, monitor their state and trigger migrations, through a web dashboard interface (appendix B).

A.2 Implementation as an Open-Source Project

As mentioned, we implemented ThingsMigrate as an open-source project (built over ThingsJS). The version of ThingsMigrate/ThingsJS that correspond to this paper has been tagged as `ecoop2018` in our GitHub repository and can be accessed at: `https://github.com/karthikp-ubc/ThingsJS/tree/ecoop2018`. Given the availability of similar hardware and software configurations, it can be used to reproduce the results that we obtained.

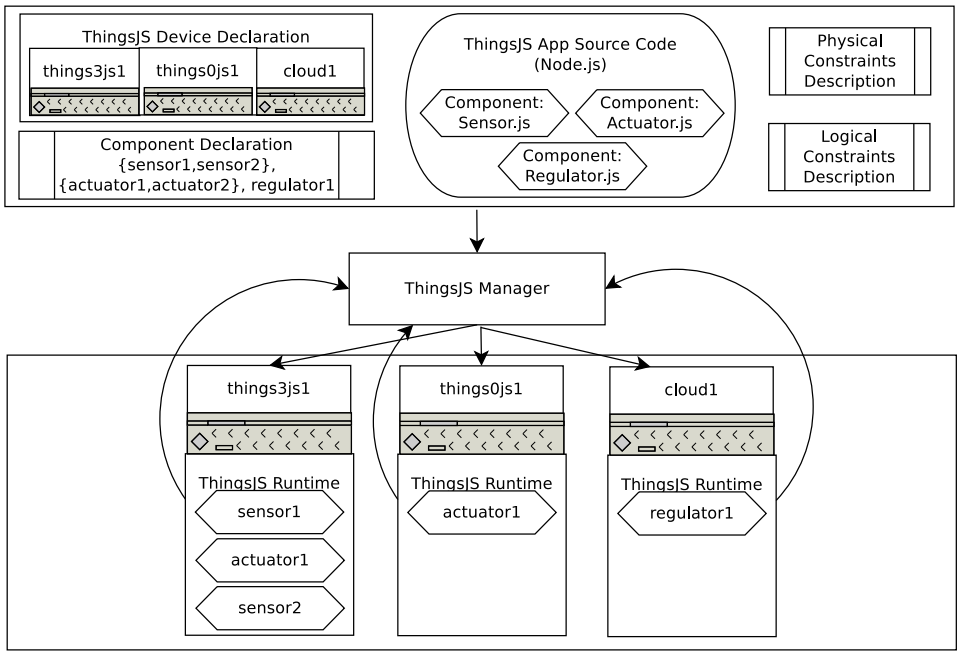


Figure 14 High-Level Architecture of ThingsJS

As ThingsJS provides a IoT-based middleware, it needs be installed and run on every device. Each device then executes a *worker* node (corresponding to the ThingsMigrate/ThingsJS Runtime in our architecture) that hosts one or more JavaScript applications. Worker nodes listen for appropriate `start` and `stop` commands over the pub/sub interface to respectively start and stop the execution of applications, as well as `migrate` commands to trigger the migration between two nodes.

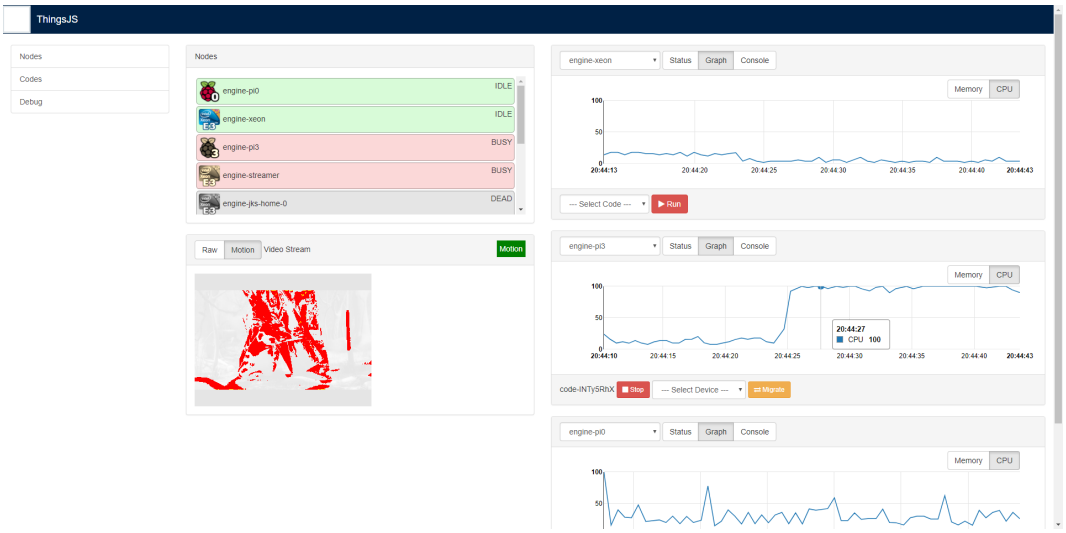
Detailed installation and usage instructions can be found at our project page[7] and in our wiki[8]. In addition, we also provide a video demo of ThingsMigrate[9], as well as a ready-to-use VirtualBox Virtual Machine image[10] that can be used to try ThingsMigrate and our web dashboard (appendix B).

We encourage the reader to try out our system – using either IoT-like devices (i.e., Rapsberry Pi, Beaglebone, etc.), or regular computers (multiple *worker* instances can be launched on the same machine to emulate additional devices).

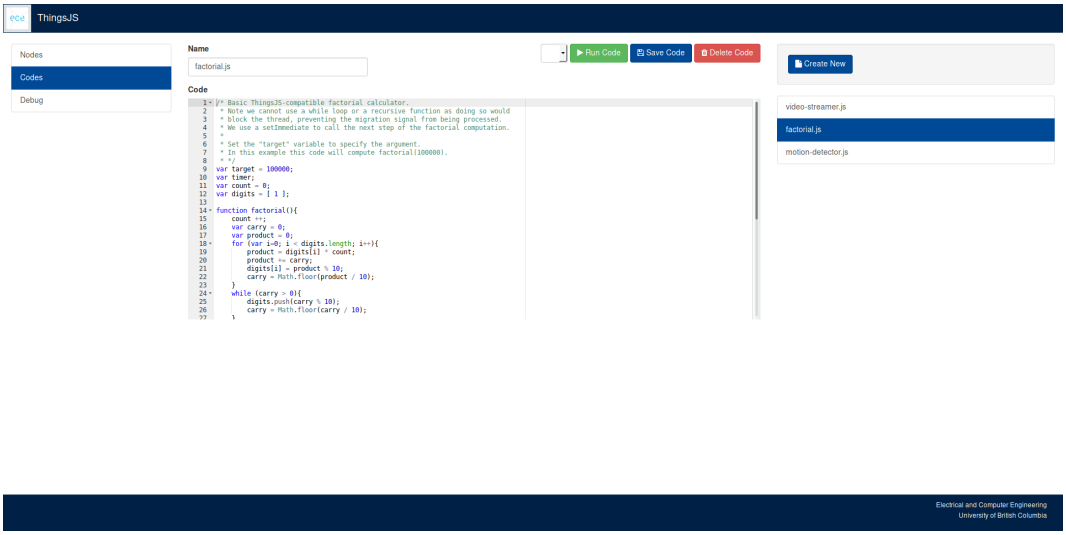
B Web Dashboard

In addition to implementing the ThingsMigrate system over ThingsJS, as well as the code instrumentation, snapshotting and code generation algorithms as a set of command-line tools, we also implemented a *Web Dashboard* as a user-friendly interface to interact with ThingsMigrate. This appendix highlight the main features of our dashboard.

[7] `https://github.com/karthikp-ubc/ThingsJS`
[8] `https://github.com/karthikp-ubc/ThingsJS/wiki`
[9] `http://ece.ubc.ca/~karthikp/ThingsMigrate/ecoop2018.html`
[10] Idem.



(a) Overview



(b) IoT Apps Source Code

Figure 15 ThingsMigrate Dashboard

B.1 Interface Overview

Our web dashboard can be used to view the status of the currently available devices and applications, and can also launch, migrate and stop the execution of applications across devices. Given that the scheduling infrastructure is not yet implemented, the dashboard plays the role of the *ThingsMigrate Manager* by allowing the the user to control the deployment and the migration of applications to IoT devices manually.

Screenshots of the dashboard are shown in Figure 15. The dashboard provides three different views, which can be selected through the menu:

1. **Main (default) view** (Figure 15a): this view provides a holistic view of all the nodes (devices – IoT or cloud-based), their detailed status, performance (CPU and memory usage) and console output. In addition, specific to the video streaming / motion detection case-study application (Section 7), this view allows one to observe the raw video stream as well as the detected motion patterns.
2. **Codes view** (Figure 15b): this view provides a code viewer and editor to view and edit the source code of the IoT apps to be run on the devices. Developers can write their apps there directly, or cut-and-paste from their favorite editor / IDE into the code editor.
3. **Debug view**: this view provides network debugging support by showing the flow of pub/sub messages across the pub/sub substrate.

B.2 Main Features

In a nutshell, our dashboard provides the following features:

- **Viewing the state of worker nodes** (*Main view*, similar to Figure 15a): in the top-left portion of the main view, a list of all registered worker nodes is shown, with their status:
  - **Green**: the node is currently available and idle (i.e., not executing any application)[11]
  - **Grey**: the node is currently available, but busy (i.e., executing another application)
  - **Red**: the node is currently unavailable
- **Viewing detailed worker status** (*Main view*): on the right side, the interface provides three panes that can be used to show detailed information on a given worker (IoT/cloud) node. Each status pane provides three different views:
  - **Status**: shows the status of the node
  - **Graph**: shows a graph of the memory or CPU usage of the node
  - **Console**: shows the output of the application that the node is currently executing
- **Launching applications on devices** (*Main view*): the dashboard can be used to launch any application defined in the `Codes` section of the dashboard. The code is *instrumented* on-the-fly by ThingsMigrate (Section 4.5) in order to support migration, and is launched through the Runtime on the target device.
- **Stopping applications** (*Main view*): the execution dashboard can instruct the Runtime on any target device to stop the execution of a given application.
- **Migrating applications between nodes** (*Main view*): the dashboard can trigger the migration of an application from one device to another. When doing so, it instructs the Runtime on the first device to initiate the migration. The Runtime will then pause the execution, take a snapshot of the current state, send the state to the Runtime of the second device, and instruct the second device to resume the execution (Figure 1).

[11] Although our framework supports executing several applications on a given worker node, our web dashboard currently allows for only one application to be mapped to a given worker node.