Poster: Towards a Distributed and Self-Adaptable Cloud-Edge Middleware

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

Kumseok Jung
Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
kumseok@ece.ubc.ca

Karthik Pattabiraman
Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
karthikp@ece.ubc.ca

***Abstract*—The Internet of Things (IoT) landscape has grown tremendously over the past few years. Modern devices are getting more powerful, and are therefore gaining the ability to execute complex and rich applications (*edge computing*), which can yield many benefits compared to traditional, cloud-centric models. On the other end, the use of high-level languages (e.g., JavaScript) allows programmers to abstract low-level considerations, and gives the ability to run the same code across different platforms. In this paper, we describe the main features of ThingsJS, our comprehensive self-adaptive cloud-edge middleware that allows for designing and running high-level, complex applications written in JavaScript on the IoT devices themselves.**

***Index Terms*—Cloud-Edge Computing, Internet of Things, Distributed Systems, Middleware, JavaScript, Self-Adaptation**

I. Introduction

The Internet of Things involves a plethora of interconnected devices that produce and consume large amounts of data. Over the last few years, the IoT landscape has grown tremendously, and applications can be found across multiple domains, both in academic and industrial circles. Gartner estimates that there will be over 20 billion IoT devices by 2020 [1].

While traditional IoT devices consist mostly of low-level embedded sensors and actuators, a more recent generation of devices has emerged. Such devices; e.g., the popular Raspberry Pi series, are getting more and more powerful and connected, and are quickly evolving beyond their traditional role of low-level embedded sensors and actuators to become more akin to *embedded computers*. As such, they can execute full operating systems, such as various distributions of Linux, thereby opening the door to running complex, rich and high-level applications *directly on these devices*, at the *edge* of the network (i.e., *edge computing*). Overall, we believe that edge computing provides many benefits, compared to the traditional (i.e., cloud-centric) model, such as reducing bandwidth and infrastructure costs; alleviating the dependence on third-party datacenters and on constant Internet connectivity; and reducing latency, which can have an impact on real-time data processing and decision making [2].

On the other hand, to bridge the heterogeneity of the IoT landscape, there is a strong incentive for establishing common paradigms and high-level abstractions. We believe this can be done through high-level, platform-independent languages and frameworks. Not confined anymore to the realms of the web, JavaScript has grown over the last few years as a mature and dynamic language on its own [3], and applications can be found in the desktop, mobile and server spaces, and more recently, in the IoT world [4]–[14]. In addition to its popularity, JavaScript has many properties (e.g., event-driven and asynchronous model) that make it well suited for developing high-level, platform-independent IoT applications.

In this paper, we present ThingsJS[1], our comprehensive platform for designing and deploying high-level edge applications written in JavaScript *onto the IoT devices themselves*, in combination with the cloud. ThingsJS provides a set of APIs and high-level services for developers, a set of self-adaptation features, and rich graphical and console-based user interfaces to observe and interact with the system.

II. ThingsJS: A Cloud-Edge Framework

1) Edge Application Model: ThingsJS primarily executes *high-level* applications *in the edge*[2], but also takes advantage of cloud resources. At its core, a ThingsJS deployment contains a set of devices (edge and cloud). Developers write *ThingsJS Applications*, which are logical packages regrouping a set of *components* (i.e., written in JavaScript) to be executed on the available devices. For instance, a typical temperature regulation application for a large building might comprise a set of sensor instances that collect temperature readings (e.g., `sensor.js`), a set of actuator instances that control the power output to the heaters / AC units (e.g., `actuator.js`), as well as one instance of a regulator (e.g., `regulator.js`) to manage the temperature across the building, based on sensor data. Other examples of ThingsJS applications include a video-surveillance application, which comprises a set of *video camera* components, and a set of *motion detection* components that detect potential intrusions [16]).

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).

[1] A vision paper outlining an earlier vision of ThingsJS was published at the 2017 Middleware for IoT (m4iot) workshop [15]. This paper presents our more recent work and vision.

[2] We use the term *in the edge* to indicate that high-level applications are running on the IoT devices themselves. Some other work use the expression *close to the edge* to refer to applications that are running on intermediate nodes of the cloud-to-device infrastructure, etc.

2) Constraints: Given the amount of devices and components that a ThingsJS system can comprise, requiring developers to manually deploy components to devices is neither feasible nor optimal. Rather, ThingsJS manages the scheduling in a completely automated fashion (more in Section III-1). To assist ThingsJS in making optimal scheduling decisions, ThingsJS provides a rich constraint system that developers use to express the capabilities and requirements of the devices and components (e.g., processing power (CPU), memory, bandwidth, latency) as well as any application-specific constraint (e.g., the required FPS rate for the video-surveillance application, etc.). Classes of devices can be specified in a hierarchical manner, allowing system operators to define constraints for multiple devices at once. Also, ThingsJS also allows for some constraints to be refreshed at *runtime* – for instance, the remaining memory on the devices, or the resources consumed by an application, can be monitored and updated.

3) APIs and Services: The ThingsJS framework exposes a rich set of APIs that developers can leverage to access a set of distributed services provided by the ThingsJS Runtime. In particular, developers can access a distributed file system shared by all nodes (built over MongoDB), can use the MQTT interface to communicate with other nodes (more below), and can interact with the various components of the ThingsJS Runtime (Section III).

4) Communications: To provide more standardized communication interfaces between components, ThingsJS uses the MQTT (topic-based publish/subscribe) protocol [17], which enjoys widespread popularity in the IoT space (and across many other applications), as it provides event-based abstractions, and enables the logical decoupling of components that produce and consume content. Service implementation is currently centralized, we are working towards a decentralized solution *in the edge*.

III. THINGSJS: A SELF-ADAPTIVE RUNTIME

The ThingsJS Runtime is a distributed substrate that is deployed on all devices that are part of the ThingsJS system. It manages the execution of all ThingsJS applications, and includes self-adaptation measures to increase the dynamicity and the dependability of the system.

1) Scheduling: At its core, a ThingsJS system contains a set of devices (edge and cloud) $\mathbb{D}$, as well as a set of applications $\mathbb{A}$ each comprising a set of components $\mathbb{C}$. For each component $C \in \mathbb{C}$ running in the system, the scheduling problem involves finding the most optimal edge (or cloud) device $D \in \mathbb{D}$ to host that component, according to an optimization function. The different device-related and component-related constraints (Section II-2) must be considered, and any eventual rebalancing should attempt to minimize the *costs* of rebalancing. Currently, we have currently implemented a simple memory-based first-fit scheduler – more refined algorithms are planned.

2) Failure Detection: The ThingsJS Runtime proactively detects imminent component failures on devices, with enough lead time as to perform eventual mitigation actions. In our case, an imminent failure triggers a rescheduling, which in turn triggers the migration (Section III-3) of the component(s) in danger of failing.

3) Dynamic Migration: We have developed a novel technique for migrating *stateful* JavaScript applications between heterogeneous devices (i.e., ThingsMigrate [16]). The ThingsJS Runtime integrates this technique as part as its dynamic adaptation strategies: whenever a new schedule is produced, stateful components (i.e., components for which the state should be preserved when migrating them) are transparently migrated between *heterogeneous* devices with the help of ThingsMigrate.

4) Distributed Data and Communications: A question arises on *where* the data used by different components hosted on different devices should be placed. ThingsJS supports the transparent redirection of data streams in components that are migrated (e.g., a file read by component C on device D_1 migrated to device D_2 will be redirected from D_1 to D_2). From a more generic perspective, we believe that the global file system itself could be distributed over the cloud-edge overlay, and be provided as a *distributed service* in the edge.

We are also looking into distributed publish/subscribe models that could benefit from the resources of the edge, as to minimize the MQTT-based communication costs and latencies.

5) Global and Unified Adaptation: As mentioned earlier, dynamic adaptation can be done in various ways: the computations, the data and the communications can be distributed onto the devices, as to benefit from the edge resources to meet optimization objectives while ensuring that constraints are met. To truly unleash the potential of a self-adaptive system, there is a need to come up with *global*, *unified* adaptation policies that will take these various dimensions into consideration simultaneously. This is one of the long-term goals of ThingsJS.

IV. THINGSJS: USER INTERFACES

1) Web Dashboard: ThingsJS provides a comprehensive web dashboard interface that allows for interacting and observing the different components of the system. The dashboard allows users to observe the state of the devices (edge and cloud), create, edit, launch, migrate and stop ThingsJS applications and components, observe the status of the system scheduler (current schedule and history of past schedules) through different graphical views, etc.

2) Console-Based Interface: In addition to a web interface, ThingsJS provides a distributed console-based interface that offers a similar set of features as the web dashboard.

V. CONCLUSION

We presented ThingsJS, a middleware that allows developers to create and execute high-level distributed applications written in JavaScript *in the edge*. ThingsJS provides a rich application and constraint model, combined with a set of powerful APIs and services that ease the development of rich IoT apps. Further, it provides a self-adaptive runtime environment that dynamically schedules the execution of the applications onto the available devices. Finally, ThingsJS provides a comprehensive web dashboard and command-line interface for administering and monitoring the system.

REFERENCES

[1] "Programming the internet of things with node.js an," https://www.gartner.com/newsroom/id/3165317, 2018.

[2] M. James, M. Chui, P. Bisson, J. Woetzel, R. Dobbs, J. Bughin, and D. Aharon, "The internet of things: Mapping the value beyond the hype," *McKinsey Global Institute*, vol. 3, 2015.

[3] "Tiobe index," https://www.tiobe.com/tiobe-index/, 2017.

[4] "Programming the internet of things with node.js and html5," https://conferences.oreilly.com/solid/internet-of-things-2015/public/schedule/detail/40797, 2018.

[5] "Nodebots – the rise of js robotics," http://nodebots.io/, 2018.

[6] "Cylon.js – javascript framework for robotics, physical computing, and the internet of things using node.js," http://cylonjs.com/, 2018.

[7] "Johnny-five: The javascript robotics & iot platform," http://johnny-five.io/, 2018.

[8] E. Gavrin, S.-J. Lee, R. Ayrapetyan, and A. Shitov, "Ultra lightweight javascript engine for internet of things," in *SPLASH Companion 2015*. New York, NY, USA: ACM, 2015, pp. 19–20.

[9] (2017) Intel xdk. [Online]. Available: https://software.intel.com/en-us/xdk

[10] *DukTape*, 2017. [Online]. Available: http://www.duktape.org/

[11] *mjs*, 2017. [Online]. Available: https://github.com/cesanta/mjs

[12] S. Tilkov and S. Vinoski, "Node. js: Using javascript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, 2010.

[13] "Chakracore javascript engine source code." https://github.com/Microsoft/ChakraCore, 2018.

[14] "Spidermonkey - mozilla — mdn," 2018.

[15] J. Gascon-Samson, M. Rafiuzzaman, and K. 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*, ser. M4IoT '17, 2017, pp. 11–16.

[16] J. Gascon-Samson, K. Jung, S. Goyal, A. Rezaiean-Asel, and K. Pattabiraman, "Thingsmigrate: Platform-independent migration of stateful javascript iot applications," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 109. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[17] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.