Demo: ThingsMigrate - Platform-Independent Live-Migration of JavaScript Processes

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

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

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

***Abstract*—Recent trends in IoT (Internet of Things) has seen increasing number of devices being shipped with full-fledged operating systems, allowing more complex and stateful applications written in high-level languages (e.g., JavaScript) to be run on the edge. The benefits of pushing computations towards the edge is that one can reduce the network costs of data transmission. Just like any other distributed system, we need to guarantee in IoT the availability of running processes, and thus need a live-migration mechanism for such programs. However, well-studied VM migration techniques are costly and impractical in IoT, due to the resource constraints and diversity of devices. In this demo paper, we present a demo of ThingsMigrate [1], a JavaScript middleware for enabling live-migration of stateful JavaScript applications in a platform-independent manner, along with a web dashboard used to monitor and control the IoT devices.**

***Index Terms*—Internet of Things, Edge Computing, Distributed Computing, JavaScript, Process Migration**

I. Motivation

In recent years, connectivity between computing devices has rapidly grown and the number of devices connected to the Internet has already overtaken the world population [2]. Current estimates expect the number to grow to tens of billions by the year 2020 [3]. In the last decade, decreasing manufacturing costs have allowed vendors to equip the IoT devices with full-fledged operating systems, capable of running more complex and stateful applications that were traditionally run on the cloud.

For many of the IoT applications, there are financial benefits of performing computation on the edge [4]–[6]. For instance, consider a video surveillance system, which consists of a video camera producing a video stream, and a cloud server running a motion detector program that consumes the video stream. Assume that the camera produces F frames per second, that each frame is B bytes long, and that the motion detector spends C clock ticks on each frame. Given N cameras, the network cost of the system is $N \times F \times B$ bytes per second, and the computation cost is $N \times F \times C$ per second. Thus the network cost grows linearly with the number of cameras, and the cloud becomes the bottleneck when $N \times F \times C$ exceeds the clock speed of the cloud server. In contrast, if we built the system so that image processing is performed on the cameras themselves, we incur minimal network cost and the computation effort is parallelized.

Given that there will be more stateful applications running on the edge devices, and that failures can occur as in any distributed system, we need to ensure their availability. Traditionally, guaranteeing availability involves process migration [7], [8]. However, typical VM migration techniques are platform-specific and thus not easily applicable in the context of a heterogeneous computer network like IoT. For instance, migrating a process from a 32-bit little-endian device to a 64-bit big-endian device would require additional transformation to be performed on the snapshot; this technique soon becomes impractical with the diversity of IoT devices. To provide a mechanism that works on a wide range of devices, we need a high-level, platform-independent migration technique. Our insight is that we need to implement this at the programming language level to achieve high efficiency. We have built such a technique for JavaScript applications [1], since the language is becoming very popular in the IoT domain [9], [10] - this is the demo's focus.

II. Design

Our goal is to provide a platform-independent mechanism to enable migration of stateful JavaScript applications. To achieve our goal, we do not modify the underlying JavaScript VM (e.g. Node.js) as we cannot assume all device vendors will allow their VM to be modified or provide support for our technique. Instead, we enable migration via code instrumentation, making our technique applicable on any ECMAScript standard-compliant VM. Any IoT device equipped with a JavaScript engine can use our instrumenter to convert regular JavaScript code into a migratable version and run it.

Challenges While the idea of capturing a program's state and reconstructing it on a different device is simple, there are some challenges when implementing it at the JavaScript code level. An arbitrary JavaScript code contains hidden program state such as closures. Local variables captured in a closure function cannot be accessed via the native reflection API in JavaScript. Furthermore, we need to be able to capture information about pending events in the event loop, which are not directly accessible from the JavaScript code.

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).

Related Work Previous efforts have shown that JavaScript processes can be migrated between browsers with reasonable overhead [11], [12]. While these papers provide important insights about migrating a JavaScript process, their techniques are not generally applicable to the IoT domain as they either depend on an external migration service, or require modifying the underlying JavaScript VM. In our work, we address these issues to achieve platform-independent migration of JavaScript processes between IoT devices.

III. Approach

We call our approach *ThingsMigrate* [1], and it is part of a larger system we are building called *ThingsJS* [13]. At a high level, we are converting a program with hidden states into a semantically equivalent program without any hidden states, and then capturing and reconstructing the program at the destination device.

Instrumentation The main purpose of instrumenting input code is to expose the hidden state and make it accessible for capturing in a snapshot. We do this by injecting a Scope object into each function to track its local state, and constructing a Scope Tree. Each node in the tree is updated whenever a reference is updated; for example, when a variable is assigned a new value. Thus we maintain an explicit up-to-date copy of the internal heap. We also intercept all the timer events to track the state of the event queue. The instrumented program is then executed as a child process of the ThingsJS runtime.

Serialization When executed, the instrumented program connects to the Pub/Sub (Publish/Subscribe) service and it can exchange messages with other devices. Upon receiving a `snapshot` command, a running process produces a snapshot of itself as a JSON string and then publishes it. The JSON snapshot is picked up by the target device for restoration.

Restoration Upon receiving a snapshot, a ThingsJS runtime traverses the Scope Tree and generates code that reconstructs the program state. All the closures and transient objects are restored at the beginning of the resumed program, and the code retains its original structure. The generated program is then run as a child process.

IV. Implementation

ThingsMigrate is written as an NPM module and can be installed via the NPM command `npm install things-js`. The module consists of an API for performing code instrumentation and restoration, a CLI (Command Line Interface) for starting a ThingsJS runtime, and a web dashboard for monitoring and controlling the devices on the network. A ThingsJS runtime, or worker, is a Node.js daemon that connects to the Pub/Sub server and waits for incoming messages. A user can send control messages through the Pub/Sub interface to run, pause, and migrate a program. All the communication happens via the Pub/Sub server, which can be started using the CLI. While the CLI provides commands to run, pause, and migrate a program across the workers in the network, the web dashboard's GUI is easier to use and will be primarily used in the demo.

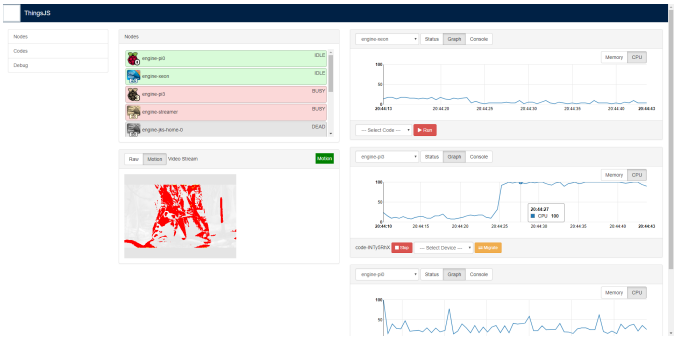

Fig. 1. Screenshot of the Web Dashboard

V. Demo Script

Using the CLI command `things-js dash`, we start the web dashboard server. By default, it binds to `localhost` and listens on port `3000`. We open the web dashboard client application on a browser. Figure 1 is a screenshot of the home page. From the home page, we can (1) view the list of devices and their status, (2) select a device to monitor its resource usage on a live graph, and (3) send commands to run, pause, and migrate JavaScript programs from one device to another. A demo video and code has been made publicly available at the Github repository for ThingsJS [1].

We set up 3 different devices: a Xeon E5 server, Raspberry Pi 3 Model B, and a Raspberry Pi Zero. Each device runs a different compilation of Node.js. On each device, we launch a ThingsJS runtime using the CLI command `things-js worker CONFIG` where `CONFIG` is a JSON file containing the Pub/Sub service URL. Upon start, the ThingsJS runtime publishes its status at topic `device-registry` to which the web client is subscribed to. The device status on the web dashboard is updated to display that the device is now active.

From the list of devices, we select the cloud server and run the `video-streamer.js` program, which reads a data stream from a video source such as a webcam or a file and publishes the stream. The web dashboard consumes the stream and displays it on the screen as a Motion JPEG. We then run the `motion-detector.js` program on the Pi 3. It consumes the stream, stores each frame in a temporary buffer, and performs image differencing on subsequent frames to detect motion. The difference images are also published as a video stream and displayed on the dashboard. Upon detecting significant change between the frames, the program publishes an alarm message. While it is running, we can observe the increase in CPU usage on the live-graph. Since `motion-detector.js` does not involve a local resource (i.e. camera), it can be run on any device. We select the Pi Zero as our migration target and then click on the migrate button. After a brief pause, Pi Zero starts running `motion-detector.js` and we observe that it continues from the last video frame processed on Pi 3. The CPU usage of Pi Zero surges up on the live graph just as it drops to 0 for Pi 3, as Pi 3 becomes idle and Pi Zero becomes busy.

[1] https://github.com/karthikp-ubc/ThingsJS

REFERENCES

[1] J. Gascon-Samson, K. Jung, S. Goyal, A. Rezaiean-Asel, and K. Pattabiraman, "ThingsMigrate: Platform-Independent Migration of Stateful JavaScript IoT Applications," in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), T. Millstein, Ed., vol. 109. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 18:1–18:33. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2018/9223

[2] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

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

[4] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[5] J. Wang, J. Pan, and F. Esposito, "Elastic urban video surveillance system using edge computing," in *Proceedings of the Workshop on Smart Internet of Things*, ser. SmartIoT '17. New York, NY, USA: ACM, 2017, pp. 7:1–7:6. [Online]. Available: http://doi.acm.org/10.1145/3132479.3132490

[6] H. Sun, X. Liang, and W. Shi, "Vu: Video usefulness and its application in large-scale video surveillance systems: An early experience," in *Proceedings of the Workshop on Smart Internet of Things*, ser. SmartIoT '17. New York, NY, USA: ACM, 2017, pp. 6:1–6:6. [Online]. Available: http://doi.acm.org/10.1145/3132479.3132485

[7] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, ser. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 273–286. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251203.1251223

[8] M. Melo, P. Maciel, J. Araujo, R. Matos, and C. Arajo, "Availability study on cloud computing environments: Live migration as a rejuvenation mechanism," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2013, pp. 1–6.

[9] 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, Nov 2010.

[10] J. Lin and K. El Gebaly, "The future of big data is... javascript?" *IEEE Internet Computing*, vol. 20, no. 5, pp. 82–88, 2016.

[11] J. T. K. Lo, E. Wohlstadter, and A. Mesbah, "Imagen: Runtime migration of browser sessions for javascript web applications," in *Proceedings of the 22Nd International Conference on World Wide Web*, ser. WWW '13. New York, NY, USA: ACM, 2013, pp. 815–826.

[12] J.-w. Kwon and S.-M. Moon, "Web application migration with closure reconstruction," in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW '17, Geneva, Switzerland, 2017, pp. 133–142.

[13] 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.