OneOS: IoT Platform based on POSIX and Actors

Kumseok Jung
University of British Columbia

Julien Gascon-Samson
ÉTS Montréal

Karthik Pattabiraman
University of British Columbia

Abstract

Recent interest in *Edge/Fog Computing* has pushed *IoT Platforms* to support a broader range of general-purpose workloads. We propose a design of an IoT Platform called OneOS, inspired by Distributed OS and micro-kernel principles, providing a single system image of the IoT network. OneOS aims to preserve the portability of applications by reusing a subset of the POSIX interface at a higher layer over a flat group of *Actors*. As a distributed middleware, OneOS achieves its goal through *evaluation context replacement*, which enables a process to run in a virtual context rather than its local context.

1 Motivation and Approach

The *Internet of Things (IoT)* is now a reality. With an increasing number of *smart* devices, the IoT topic of the year 2018 was *IoT Platform* [13]. A recent interest in *Edge/Fog Computing* (i.e., the concept of taking the workload from the *cloud* and spreading it across the IoT network) has posed new challenges for IoT platforms to evolve into more general-purpose systems, providing the flexibility to deploy arbitrary programs on a broad range of devices. Therefore, we need to design an IoT platform that can fully utilize the network's compute resources for general-purpose workloads in addition to the cyber-physical workloads. There are two high-level goals of a general-purpose IoT platform: ① to provide a dependable computing infrastructure, ② to provide a programming environment for a user (e.g., an application developer) to leverage the distributed computing features of the platform.

Building and maintaining a coherent software infrastructure for an ever-increasing diversity of devices is a massive effort. Platform-specific details need to be under strict bookkeeping, and different programming abstractions and languages need to be reconciled. Traditional reliability techniques become inadmissible due to the heterogeneity and the resource constraints of the devices. Furthermore, the solution must be future-proof to account for the long life-cycle of IoT devices. An IoT platform must meet these challenges gracefully and reduce the complexity that a user has to deal with.

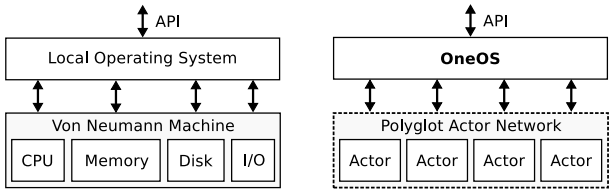

Figure 1: *OneOS* operates over a network of abstract *Actor*s, as opposed to a bare-metal Von Neumann Machine

We observe that operationally, an IoT platform is analogous to an Operating System (OS); it provides common services and I/O interfaces for user programs, and runs programs under a schedule and policy. However, unlike a Host OS, which manages the local hardware resources like CPU and memory, an IoT platform typically manages a heterogeneous network of self-contained runtime systems. Thus we propose a design of an IoT platform, which we call *Overlay Network Operating System (OneOS)*, operating in the *application layer* over a flat group of abstract *Actor*s [21] to provide a single system image of the computer network. As the model of the underlying machinery does not directly involve low-level resources, the focus of *OneOS* differs from a Host OS. *OneOS* addresses higher-level concerns regarding distributed computing: scheduling of programs on different runtime systems, coordinating inter-process communication (IPC), providing a mechanism for locating various resources within the network, storage of data, and providing an API to the user.

Existing IoT platforms and other distributed computing frameworks provide the aforementioned functionalities at the cost of *programmability*; they often require the user to use a specific API or adopt a certain programming paradigm. Alternatively, we adopt the POSIX [1] interface, which already provides abstractions sufficient for building a distributed system. Adopting a living standard that has stood the test of time can make it easier to maintain the system design and preserve the portability of applications. Previous research on POSIX

kernels have shown challenges in managing low-level hardware resources during run-time [33] and when writing the kernel itself [9]. We avoid these issues by working with high-level language runtimes, which handle the low-level tasks like garbage collection, and selecting a subset of POSIX interfaces relevant to distributed computing. By mapping abstractions like I/O streams over an *Actor* network, we can run existing programs without modification. Reading a value from a sensor and writing it to a file should be as simple as a single line of command: `cat /dev/sensor1 > sensor1.log`. To actualize this concept, we highlight the following insight: all we need to do is to adjust the *operational semantics* of a program execution – we must interpret a program in a distributed context, rather than in the local context.

2 Related Work

We first examine different approaches to building a distributed computing framework, keeping in mind 3 qualitative criteria for evaluating their usefulness: *programmability*, *maintainability*, and *efficiency*. *Programmability* describes how easy it is to write an application for the target platform, and it affects the overall productivity of the user. For instance, being able to use existing code is easier than having to write new code in a domain-specific language. *Maintainability* describes how easy it is to maintain the platform, and ultimately affects the *reliability* of the system. Many factors can affect the *maintainability*, such as the need to add redundancy, heterogeneity of the software stack, and the complexity of the network topology. Finally, *efficiency* describes the runtime performance of the platform, and it can be measured more concretely in terms of memory and bandwidth usage. The 3 criteria can be quantified as *cost of development*, *cost of maintenance*, and *cost of computation* respectively. However, as we do not yet have an evaluation framework for speaking in terms of *cost*, we discuss them qualitatively throughout this paper.

IoT Middlewares. Without project-specific details, we first discuss the general organization of an IoT middleware. We observe that most systems [15, 17, 25, 32, 35] are made up of a set of services, organized in a hierarchy. There is a loose coupling between the logical software topology and the physical network topology. For instance, a local "manager" service aggregates data from "worker" services in the local area network, then communicates with a "compute service" on the *cloud*. While there is nothing fundamentally preventing a user from deploying the "manager" service on the *cloud*, the platform design assumes that the logical structure mirrors the physical layout; breaking this assumption may render the system to operate sub-optimally, or even be unusable. As there are operational dependencies between the infrastructural components, maintaining the whole system is not trivial. We also observe that most systems are designed for specific use cases and are not general-purpose. The user must write applications specifically for the target framework, using specific APIs and following a certain programming paradigm [5, 11, 14, 30].

Cloud Management Platforms. *Cloud Management Platforms* are generally organized in a flat cluster topology, which is easier to maintain and scale [4, 22, 23]. Such systems are useful for scaling applications that are laid out as a set of micro-services, which are *embarrassingly parallel* [26]. For more complex deployment scenarios in which the services have operational dependencies, the user is responsible for configuring the platform accordingly. We note that these platforms are designed and optimized for horizontal scaling of containerized applications, and they are not designed for providing a general-purpose application platform. As we target IoT systems, which are geographically spread out and resource-constrained, Cloud Management Platforms are not directly applicable for our use case.

Distributed Operating Systems. Distributed OSes [6, 10, 29, 38] provide a single system image of a network of computers. The complexities of managing heterogeneous set of resources are hidden under an abstraction layer, and the user simply interacts with a single interface. Philosophically, this concept is closest to the system we envision. However, previous research has identified several challenges such as distributed shared memory, clock synchronization, and context-switching, making it diffcult to achieve a practical implementation and adoption of this design [36, 37]. While there has been significant work in addressing some of these problems [6, 27], building OSes from scratch and keeping up with the pace of the IoT landscape requires a monumental investment.

3 System Design

One of our design goals is to provide a single system image of the computer network by hiding the logistical complexities of distributed computing. At the same time, we want to minimize the effort a user needs to make to write an application. Ideally, we want to be able to run existing programs transparently without modification. As we target the IoT ecosystem, we make the following assumptions: ① the machines in the network have different processor architectures and memory layouts, and each run a POSIX-based Host OS, even if they are relatively constrained (e.g., Raspbian OS on Raspberry Pi Zero with 1GHz single-core CPU and 512MB RAM). We do not include microprocessors in our model; they are treated as peripheral devices. ② All applications that run on *OneOS* are written in a high-level language like JavaScript or Python. Thus, we require that each machine has at least one high-level language runtime, such as Node.js or CPython. ③ We do not assume any particular physical network topology. ④ The majority of devices are unmonitored and may be deployed for a long time.

Keeping in mind our design goal and the assumptions, we discuss our model of the platform and articulate the design choices we make.

3.1 Logical Network Topology

OneOS is a thin virtual layer realized by a network of middleware services, which we refer to as *OneOS runtime*s, organized as a flat cluster. Instead of a structured, hierarchical logical topology, we adopt the flat topology because it is more resilient against arbitrary node failures [39]. Furthermore, in contrast to a hierarchical design, there is no logical coupling between the physical structure of the network and the software model [40], making it easier to maintain and deploy. Coordinating communication in a flat software layout over a geographically dispersed network (i.e., building a *grid* [12]) can be challenging due to the dynamic and heterogeneous nature of the network. We address some of the challenges by using the Publish-Subscribe (Pub-Sub) interface within the underlying communication infrastructure. The Pub-Sub interface provides an extra level of indirection between the communication endpoints via software-defined topics, which allows for communication logic free from low-level network properties like IP addresses. By representing network resources in the form of topics, *OneOS* is able to provide a dynamic and topology-agnostic communication infrastructure. While the Pub-Sub interface simplifies the programming interface, it incurs performance overhead, since all packets go through a broker and consume additional bandwidth. Some overhead can be relieved by using the Pub-Sub infrastructure only for exchanging metadata such as public IP addresses before switching to direct TCP communication. However, such an optimization has not been done in the current implementation.

3.2 Middleware Model

Drawing some parallels with Distributed OSes, the *OneOS runtime* is analogous to a distributed kernel. Unlike a kernel, the *runtime* operates high-level language runtimes – such as Node.js or CPython – via message passing. As a comparison, consider a scenario where we execute a program. A traditional kernel is responsible for reading the program from a disk, allocating memory for the program, initializing the links and the environment, and scheduling the processor. Thus, a kernel operates the hardware resources directly. In contrast, the *runtime* is responsible for creating a virtual evaluation context for a program, then executing the program with the appropriate high-level language runtime. A *runtime* is thus modeled as an *Actor* that can create another *Actor*.

As a middleware-based platform, *OneOS* is limited from having granular control over the hardware resources such as memory and thread. The smallest unit of computation in *OneOS* is a process, which we call an *agent*, and not a single CPU instruction. An *agent* is treated as a black-box with inputs and outputs, thus is again modeled as an *Actor*. This coarse-grained control of resources is a desired property. Previous work on Distributed OS has demonstrated the challenges in dealing with low-level resources over the network [3, 6, 31, 33]. As the network latency would be orders of magnitude larger in an IoT setting, generally it would be more efficient to run a single process on a single host rather than splitting it across a network.

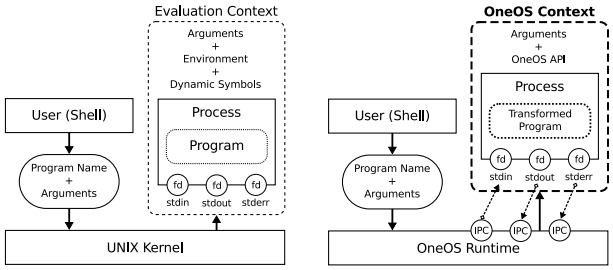

Figure 2: Similar to a UNIX kernel, a *runtime* is responsible for initializing a runtime environment. A *OneOS* context encapsulates the process to evaluate the given program in a distributed environment, redirecting all the system calls to the appropriate *kernel agent* on the network.

Evaluation Context Replacement. If a *runtime* naively spawned an *agent* as a child process, it runs as a regular UNIX process accessing local host resources. In order to interpret the program in the *OneOS* virtual context, the *runtime* performs an *evaluation context replacement* procedure before it spawns the program as a child process. The purpose of this procedure is to intercept all the system calls and redirect them through the network to the respective system-level services, referred to as *kernel agent*s, thus changing the run-time semantics and allowing the program to run in a virtual context distributed over multiple *runtime*s. The procedure comprises 2 phases: ① an offline code instrumentation step, performed once per program, and ② an environment bootstrapping step carried out just before instantiating a child process.

During the code instrumentation step, all the built-in modules abstracting the POSIX API (e.g., `fs` in Node.js) are replaced with the equivalent *OneOS* API. For example, consider the following JavaScript program:

```
1 var outpath = process.argv[2];
2 var fs = require('fs');
3 process.stdin.on('data', (data)=> {
4   fs.appendFile(outpath, data, (err)=> {});
5 });
```

Code 1: `logger.js`

This simple program expects a file path as an argument, receives data via `stdin`, and it appends the received data to the specified file. After instrumentation, the native `fs` module is replaced with `oneos/fs`. As a result, the `appendFile` call does not use the underlying kernel API, but instead it gets routed via the network to the File System *agent*. Since we have limited our application space to high-level languages

(e.g., JavaScript), the overhead of code instrumentation is relatively low due to a much smaller code footprint compared to the same logic in a low-level language (e.g., C). Moreover, the POSIX API is already abstracted as built-in modules in high-level languages, and the instrumentation step is just replacing import statements.

Taking the instrumented code, a *runtime* then creates a new environment before spawning it as a child process, injecting environment variables that represent the virtual *OneOS* environment. The child process is assigned a globally unique *agent* ID, different from the PID assigned by the Host OS. Additionally, the *runtime* creates unique Pub-Sub streams and routes them to the corresponding I/O streams of the child process, enabling IPC between processes over the network. `process.stdin` in Code 1 receives data via a Pub-Sub channel `{AgentID}/stdin` instead of the native FD 0 of the UNIX process. This environment bootstrapping step introduces a small delay before the execution of a program, but does not incur overhead during run-time. However, network queuing delay can propagate to the I/O streams of the child process.

3.3 Service Model

Having discussed the infrastructure, we now discuss the service model, which enables the system to serve as an OS for an IoT network. Amongst the *runtime*s, some of them are selected to be *kernel runtime*s, based on hardware capacity and physical location. These *kernel runtime*s form a quorum, which collectively decides which *kernel agent*s to run on which *kernel runtime*, using a consensus protocol like Paxos [24] or Raft [28]; the exact consensus protocol is an implementation detail and not part of the *OneOS* design. A *kernel agent* performs a single root-level service expected to be provided by an OS, and thus *OneOS* adopts the microkernel design. We describe below the essential services.

Inter-Process Communication Service. The IPC *agent* mediates the communication between different *agent*s, and is implemented as a Pub-Sub service. For example, when piping data between *agent*s via a command such as `./foo.js | ./bar.js`, the *runtime* that executes `foo.js` publishes the output of `foo.js` to the IPC *agent*, and the *runtime* hosting `bar.js` subscribes to that data stream.

File System Service. Adopting the UNIX philosophy that "everything is a file", the File System *agent* provides an indexing mechanism for locating resources within the network including regular files, I/O of peripheral devices, and network sockets. We decouple the indexing mechanism from the monolithic file system and define a separate storage service.

Storage Service. The Storage *agent* is responsible for storing blocks of data in the appropriate storage devices. As a *kernel agent*, it just needs to provide a consistent read-write interface to other *agent*s, and the underlying storage mechanism can be provided by exiting storage services. The current implementation uses a central database.

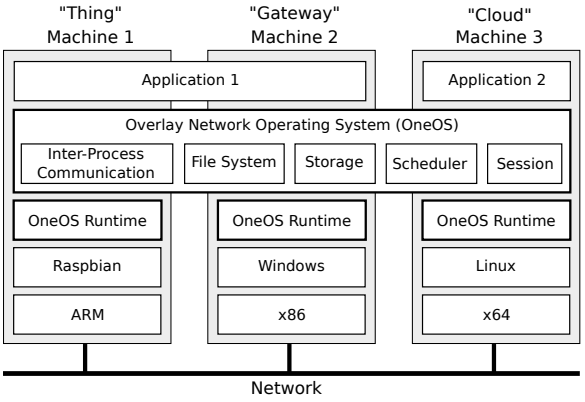

Figure 3: *OneOS* is organized as a middleware-based distributed operating system

Scheduler Service. The Scheduler *agent* is responsible for deciding where to deploy a new *agent* upon request. Upon receiving an execution request from another *agent*, such as the Session *agent*, it decides from the list of *runtime*s where to run it. Once it has decided where to run the new *agent*, it sends a command message to the target *runtime*. The current implementation uses a greedy first-fit algorithm, but a more appropriate scheduling algorithm is an area of active research.

Session Service. Finally, the Session *agent* manages interaction with a user. A user contacts the session *agent* to authenticate and create a session. Upon signing in, the session *agent* creates a new Shell *agent* for the user, and the user can interact with the system through the *Shell Client* or through a graphical interface via the *Web Client*. Any browser can be used as a graphical terminal for the system.

3.4 Operation Model

We now describe the operation of the platform to provide an intuition for applying this design in practice. When deploying the platform, the *runtime* middleware is installed on each device. Upon installation, 2 pieces of information are configured on the device: ① an RSA keypair, representing the identity of the *runtime*, and ② the location of a Name Server, which serves the *boot record* containing information about the *kernel runtime* quorum and the list of start-up *kernel agent*s. The public key of the *runtime* must be registered in the Name Server for it to be recognized as part of the cluster. Upon starting up, a *runtime* contacts the Name Server and fetches the *boot record*, and then announces its membership. If it discovers its role as a *kernel runtime*, it joins the consensus quorum. The quorum ensures that all the *kernel agent*s defined in the *boot record* are running. Thus, a *runtime* dynamically loads its behaviour from the Name Server, making it easy to update the platform; only the *boot record* needs to be updated and a *runtime* simply needs to restart.

3.5 Programming Model

Finally, we provide a simple example to demonstrate how POSIX abstractions map to the distributed context, enabling the use of regular JavaScript programs. Consider the following 3 programs `sensor.js`, `controller.js`, and `actuator.js`, together constituting an IoT application.

```
1 var GPIO = require('onoff').GPIO;
2 var sensor = new GPIO(4, 'in');
3 sensor.watch((err, data)=>{
4   process.stdout.write(Buffer.from([data]));
5 });
```

Code 2: `sensor.js`

```
1 process.stdin.on('data', (data)=>{
2   if (data[0] === 1)
        process.stdout.write('ON');
3 });
```

Code 3: `controller.js`

```
1 var GPIO = require('onoff').GPIO;
2 var actuator = new GPIO(7, 'out');
3 process.stdin.on('data', (data)=>{
4   if (String(data) === 'ON')
        actuator.write(1, (err)=>{});
5 });
```

Code 4: `actuator.js`

`sensor.js` writes to `stdout` whenever the value on GPIO pin 4 changes. `controller.js` receives data via `stdin`, and outputs the string `ON` when the input value is 1. `actuator.js` writes 1 to GPIO pin 7 whenever it receives the message `ON` via `stdin`. These are generic JavaScript programs, using only the `onoff` module [8] commonly used for interfacing GPIO devices. Now, consider the following shell command:
`./sensor.js | ./controller.js | ./actuator.js`

The Scheduler deploys `sensor.js` and `actuator.js` on *runtime*s with GPIO, while `controller.js` can be deployed on any *runtime*. The *runtime*s transparently create IPC pipes between the 3 processes over the network, as `stdin` and `stdout` objects are replaced with the *OneOS* stream objects.

4 Implementation

We have written a prototype of *OneOS* entirely in JavaScript. While the choice of language is an implementation detail, we chose JavaScript for the following reasons: JavaScript is the language of the Web, and the Web is a major part of the IoT landscape (i.e., the Web of Things [19]). Since the advent of Node.js, the server-side codebase has grown substantially and many libraries exist [7, 18] for interfacing physical hardware. We reduce the development overhead by adopting this large codebase. Moreover, its single-threaded execution and event-driven model makes it trivial to build *Actor* systems. Although the *runtime* itself is written in JavaScript, it currently supports running JavaScript, Python, and WebAssembly programs. We envision that supporting WebAssembly opens door to other languages that can be compiled to WebAssembly [20].

5 Evaluation

To evaluate our design, we consider the 3 criteria we discussed in section 2. As we do not have a quantitative evaluation framework at the moment, we only suggest a plan and provide preliminary speculations. Evaluating *programmability* and *maintainability* will involve user studies in which we compare the experience of developers when using well-known platforms versus *OneOS*. For evaluating the *efficiency* of our design, we make reasonable assumptions about the target applications and use *Distributed Stream Processing Systems* as an evaluation testbed – these systems are used for processing data streams, such as sensor data produced by IoT applications [34], and can be deployed in the *edge* as *services* [2, 16].

As *OneOS* is a general-purpose framework, we do not expect it to outperform specialized software systems. However, we can define a reasonable target for deciding if its performance is acceptable. Intuitively, we can say that the platform is "good enough" if it does not introduce a significant run-time overhead. More precisely, if running a *service* on *OneOS* does not incur a large enough overhead to cause a back-pressure in the stream-processing pipeline, it is acceptable. For instance, consider a *stream operator* receiving data frames at N frames per second (fps). To avoid back-pressure, it must process each frame within $\frac{1}{N}$ seconds. Assuming *OneOS* adds a run-time overhead of $100D\%$, we can define a *maximum frame processing interval* of $\frac{1}{N(1+D)}$. The real processing time varies for every *runtime-service* pair, so we model the *real processing interval* as a function $p(r,s)$ of *runtime* r and *service* s. If *OneOS* distributes the set of *services* among the *runtime*s, such that $\forall\langle r,s\rangle, p(r,s) \leq \frac{1}{N(1+D)}$, the performance is acceptable. Thus, the *efficiency* of the system is heavily determined by the scheduling policy, which is a part of our research plan.

Acknowledgments

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

Availability

A proof-of-concept of *OneOS* is available on Github at: https://github.com/DependableSystemsLab/OneOS.

6 Discussion Topics

As our research is at the intersection of several domains such as Computer Systems Organization, Software Engineering, and Theory of Computation, we look forward to receiving feedback from researchers from various backgrounds. In particular, we welcome comments on: modeling an abstract machine as an *Actor* network, suitability of high-level languages for systems programming, and methodologies for evaluating programming platforms and frameworks.

One of our main design choices is the adoption of the POSIX interface. POSIX is heavily tied to the C standards, which is based on a sequential abstract machine. What we have attempted is mapping a subset of the POSIX API that are not tightly-coupled with C semantics (e.g., file system, streams, sockets, etc.), over an alternative abstract machine with concurrent runtimes. Since our model anticipates concurrency and reliability issues due to its networked nature, it might be difficult to conform to POSIX specifications.

The major part of our work is about the rationale behind the design choices and the trade-offs we make in building a distributed software platform. Hence, we expect to see discussions about competing ideas in system organization, OS architecture, programming practices, etc. Some examples are: distributed versus centralized layout, peer-to-peer versus server-client communication, monolithic kernel versus micro-kernel, single-system image versus explicit system image, and the trade-off between run-time efficiency and programmability.

As this is early-stage work, we have not addressed many important parts of the design, such as service scheduling. The scheduling problem is further complicated by the fact that there are additional constraints like locality of cyber-physical resources. Certain programs need to be "location-aware" and POSIX falls short in providing a useful abstraction. Thus, there is a need for a semantics that can model this aspect, and provide abstractions that a scheduling algorithm can incorporate.

We have situated our platform entirely in the application layer by restricting the programming space to high-level languages. The reason for this is because it is difficult to dynamically replace the evaluation context of a compiled binary (e.g., a statically linked executable) due to the heterogeneity of host platforms. By making this choice, we have essentially lost the ability to directly control the bare-metal. Would there be serious limitations with this approach, such as not being able to guarantee certain security properties? The impact on performance is another concern, as we do not have control over low-level optimizations. How far can we optimize at this higher and abstract layer, and will it yield acceptable performance?

References

[1] Ieee standard for information technology–portable operating system interface (posix(r)) base specifications, issue 7. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pages 1–3951, Jan 2018.

[2] Gayashan Amarasinghe, Marcos D de Assunçao, Aaron Harwood, and Shanika Karunasekera. A data stream processing optimisation framework for edge computing applications. In *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pages 91–98. IEEE, 2018.

[3] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, , R. Rajamony, , and W. Zwaenepoel. Treadmarks: shared memory computing on networks of workstations. *Computer*, 29(2):18–28, Feb 1996.

[4] The Kubernetes Authors. Kubernetes. https://kubernetes.io/docs/concepts/. Accessed: 2019-03-10.

[5] The ThingsBoard Authors. Thingsboard. https://github.com/thingsboard/thingsboard/, jan 2019.

[6] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[7] The Johnny-Five Contributors. Johnny-five. http://johnny-five.io/. Accessed: 2019-03-10.

[8] Brian Cooke. onoff. https://github.com/fivdi/onoff. Accessed: 2019-03-10.

[9] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 89–105, Carlsbad, CA, 2018. USENIX Association.

[10] Sean M Dorward, Rob Pike, David Leo Presotto, Dennis M Ritchie, Howard W Trickey, and Philip Winterbottom. The inferno operating system. *Bell Labs Technical Journal*, 2(1):5–18, 1997.

[11] Jeffrey M. Fischer and Rupak Majumdar. Programming event processors with thingflow. In *ICCPS '19: Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, pages 67–76, New York, NY, USA, 2019. ACM.

[12] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[13] Eclipse Foundation. Iot developer survey 2018. https://www.slideshare.net/kartben/iot-developer-survey-2018, jan 2018.

[14] JS Foundation. Node-red. https://nodered.org/docs/, jan 2019.

[15] 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, New York, NY, USA, 2017. ACM.

[16] Julien Gedeon, Michael Stein, Lin Wang, and Max Muehlhaeuser. On scalable in-network operator placement for edge computing. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE, 2018.

[17] Google. Google cloud iot. https://cloud.google.com/solutions/iot/, jan 2019.

[18] The Hybrid Group. Cyclon.js. https://cylonjs.com/documentation/. Accessed: 2019-03-10.

[19] Dominique Guinard and Vlad Trifa. Towards the web of things: Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain*, volume 15, 2009.

[20] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, IoTDI '19, pages 225–236, New York, NY, USA, 2019. ACM.

[21] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[22] Docker Inc. Docker. https://docs.docker.com/engine/swarm/. Accessed: 2019-03-10.

[23] Mesosphere Inc. Mesosphere dc/os. https://docs.mesosphere.com/1.12/overview/. Accessed: 2019-03-10.

[24] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[25] Microsoft. Azure iot edge. https://docs.microsoft.com/en-us/azure/iot-edge/, jan 2019.

[26] Cleve Moler. Matrix computation on distributed memory multiprocessors. *Hypercube Multiprocessors*, 86(181-195):31, 1986.

[27] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 221–234, New York, NY, USA, 2009. ACM.

[28] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[29] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from bell labs. *Computing systems*, 8(2):221–254, 1995.

[30] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke. Middleware for internet of things: A survey. *IEEE Internet of Things Journal*, 3(1):70–95, Feb 2016.

[31] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, volume 27, 2008.

[32] Amazon Web Services. Iot greengrass. https://docs.aws.amazon.com/greengrass/, jan 2019.

[33] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, 2018. USENIX Association.

[34] Anshu Shukla and Yogesh Simmhan. Benchmarking distributed stream processing platforms for iot applications. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 90–106. Springer, 2016.

[35] SiteWhere. Sitewhere. https://github.com/sitewhere/sitewhere/, jan 2019.

[36] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[37] Andrew S Tanenbaum, M Frans Kaashoek, Robbert van Renesse, and Henri E Bal. The amoeba distributed operating system — a status report. *Computer Communications*, 14(6):324 – 335, 1991.

[38] Andrew S. Tanenbaum and Sape J. Mullender. An overview of the amoeba distributed operating system. *SIGOPS Oper. Syst. Rev.*, 15(3):51–64, July 1981.

[39] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*, pages 329–330. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[40] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*, pages 33–67. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.