Towards a Programmable World: Lua-based Dynamic Local Orchestration of Networked Microcontrollers

Fiona Guerin
Technical University of Munich
fiona.guerin@tum.de

Teemu Kärkkäinen
Technical University of Munich
kaerkkae@in.tum.de

Jörg Ott
Technical University of Munich
ott@in.tum.de

ABSTRACT

Microcontrollers execute much of the logic that makes the appliances and infrastructure around us work. Recent years have seen a significant increase in the microcontroller capabilities, with modern designs including multiple 32-bit processor cores and integrated wireless communications. At the same time, microcontrollers still typically execute fixed code burned into their firmware, leaving much of their capacity unused. In this paper we propose a Lua-based framework for microcontrollers, capable of dynamically receiving code for execution from nearby devices via wireless networks. We show that this framework allows multiple nearby microcontroller devices to be dynamically orchestrated to compose complex services. This serves as a step towards making the physical world around us dynamically programmable, enabling the creation of new local and pervasive applications that are deeply integrated into the physical world.

CCS CONCEPTS

• **Networks → Programmable networks**.

KEYWORDS

opportunistic computing; microcontrollers; Lua

ACM Reference Format:
Fiona Guerin, Teemu Kärkkäinen, and Jörg Ott. 2019. Towards a Programmable World: Lua-based Dynamic Local Orchestration of Networked Microcontrollers. In *14th Workshop on Challenged Networks (CHANTS'19), October 25, 2019, Los Cabos, Mexico.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3349625.3355441

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CHANTS'19, October 25, 2019, Los Cabos, Mexico
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6933-6/19/10…$15.00
https://doi.org/10.1145/3349625.3355441

1 INTRODUCTION

Microcontrollers have proliferated in the past decades and are controlling virtually all the active appliances and infrastructure surrounding us today. Around 19 billion general purpose microcontrollers, costing less than $1 on average, were shipped in 2018 with double digit percentage of yearly growth forecast for at least the next four years [7]. At the same time the capabilities of these chips are increasing, with recent designs such as the Espressif ESP32[1] combining significant processing power (dual core 32-bit processor capable of 600 MIPS) with wireless connectivity (Wi-Fi and Bluetooth) and low cost ($3). Because processing power and wireless capabilities are increasing, these devices can now provide not only static but also dynamic services to their surrounding users. These devices are embedded in our world and directly control it. By dynamically orchestrating them into local distributed systems, we create hyperlocal services that physically shape our environments to serve new and better.

The first step towards realizing this goal—which we take in this paper—is to provide a dynamic execution environment for microcontrollers that can receive and execute arbitrary code and commands from nearby nodes. We focus on environments where connecting every microcontroller to the Internet is not possible or desirable, assuming only direct local device-to-device interactions. This also has desirable properties for security (only a local attack surface) and resource sharing (only between physically co-located users).

In this paper, we present a design, implementation and evaluation of a Lua-based dynamic execution environment, which can receive arbitrary Lua scripts via Wi-Fi and execute them using the Lua virtual machine. We show that this approach can be used to dynamically orchestrate a heterogeneous set of local resources—microcontrollers with different sensors and actuators—into coherent distributed services, under the control of a mobile device (e.g., a smartphone carried by a user). This corresponds to use cases where a user enters a work space, hotel room, car, etc. and has their mobile device automatically orchestrate various appliances and infrastructure in the environment to behave as customized.

[1] www.espressif.com/en/products/hardware/esp32/overview

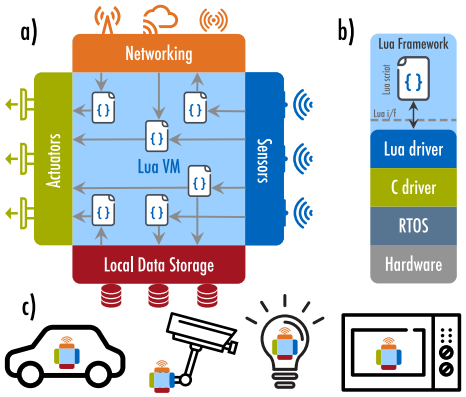

Figure 1: The execution environment for Lua scripts.

2 SYSTEM DESIGN

The goal of our framework is to enable microcontroller-powered devices to dynamically receive and execute code from nearby users. The basic idea is simple: Instead of executing the application logic as pre-installed binaries, the microcontroller uses the Lua process virtual machine (VM) to execute Lua scripts received from the environment (typically via a local wireless network, e.g., a Wi-Fi access point created by the microcontroller). First, we explain how a single platform instance operates, i.e., how it handles Lua scripts. Second, we show that small, local distributed systems can be built by orchestrating multiple platform instances.

2.1 Operation of a Single Platform Instance

To execute scripts, the system needs to provide an execution environment, which has two key functional components: 1) a **loader**, which receives scripts, makes admission control decisions and instantiates them in the VM, and 2) a **runtime system**, which executes the scripts and provides them access to the hardware and software resources of the local device.

The structure of a single instance of the Lua execution platform ("node") is shown in Figure 1a. It comprises various resources—shown around the edges of the node—including sensor and actuator hardware connected to the microcontroller, local and infrastructure based networking technologies, and local data storage. In the middle of the node is the Lua virtual machine (Lua VM) that executes dynamically loaded scripts, which can access the various resources of the device. As illustrated in Figure 1c, the microcontrollers running the framework are envisioned to be embedded in various appliances and infrastructure around the users.

This approach allows for various edge and local computing use cases to be realized simply by writing light-weight Lua scripts: **Data collection** tasks can be created via scripts that continuously read (either in push or pull mode) from local sensors and send the values to the network (either local or the cloud). Additionally, the script can buffer measurements in the local data storage to guard against data loss due to network disruptions. Simple **edge analytics** tasks can be implemented similarly to data collection scripts, except the script can execute the desired analytics computations directly in the microcontroller and send back only the processed results. Tasks can also implement **industrial or home automation** by, e.g., triggering actuators based on a schedule stored in the local data storage. Furthermore, **remote control** of the device can be achieved via scripts that read commands from the network and pass them to local actuators. Since the tasks are executing locally with a minimal latency, **feedback control logic**—e.g., a proportional-integral-derivative (PID) controller—can be implemented as a simple Lua loop that reads a sensor value, calculates the error term and the resulting control variable and then invokes the actuators to take corrective action. Further, the setpoint of the controller could come from the network (centralized cloud backend and/or a local mobile device), a local user interface (read as a sensor) or from a static configuration.

Loader: Before the scripts can be executed they must be loaded into the system. The first step of the loader is to receive and deserialize a script and the associated metadata from a network connection. The script itself is transmitted as a string (which will eventually be fed to the Lua VM) and the metadata as structured data (e.g., list of typed key-value pairs that specify parameters such as the required hardware). After getting the script and its metadata, the loader will decide whether to immediately load the script into the Lua VM, queue it for later execution, or discard it entirely—this decision may be based on a number of factors. The first factor are the dependencies: Each script may have specific hardware and software dependencies, which are included in the metadata and the loader will evaluate whether the device fulfills them. Finally, the loader will consider the current load and resource utilization, as well as any priority parameters in the script metadata, to decide whether there are currently enough resources available for execution. The loader can then decide to immediately schedule the script for execution and hand it to the runtime system, or it may queue it for later execution when more resources become available.

Runtime system: The actual execution of the scripts is done by the Lua VM along with supporting framework functionality, such as sensor/actuator drivers—which together comprise the runtime system. The resources—sensors, actuators, data storage and networking—are exposed to the scripts via well-defined Lua interfaces, as shown in the Figure 1b. We assume the hardware access is virtualized by an operating system (e.g., FreeRTOS in our implementation), which provides a C interface to the various capabilities of the hardware. These capabilities are exposed to Lua scripts

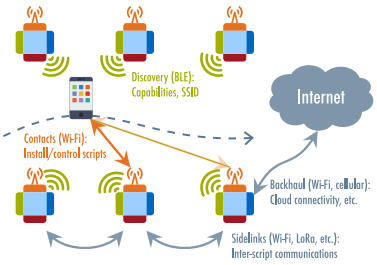

Figure 2: A pervasive system composed of communicating platform instances with a phone as an orchestrator.

by drivers in the Lua framework. The driver is composed of a C-half and Lua-half—the former uses the C interface of the OS and adapts it for use in the Lua VM via Lua's C-interworking features (the well-developed Lua C API is a major reason to use Lua for the framework). The framework further abstracts the resource access to either *push* or *pull* based mode, where pull-based operations simply call the C functions, while push based operations trigger Lua callbacks from the C-code (e.g., if the resource is interrupt driven).

2.2 Orchestrating Multiple Platform Instances

While a single platform instance enables the dynamic programmability of a single device, the real value of the approach comes from the ability to coordinate the behavior of multiple nearby instances, i.e., to orchestrate pervasive services. We assume an unstructured model—in contrast to many structured edge-computing systems—where each platform instance is deployed and acts independently without any central control systems. Instead, the orchestration is performed by nearby mobile nodes, which scan the environment for platform instances and install various scripts in them that together comprise the pervasive service.

The structure of a pervasive system is shown in Fig. 2, where the orchestrating mobile device at the center is moving in an environment with multiple platform instances with heterogeneous sensors, actuators and other resources. Conceptually we have three types of network functions: 1) **discovery** of nearby nodes and their characteristics (green in the figure), 2) **contacts** used to install scripts from the mobile to the nodes (orange), and 3) **backhaul** and **sidelinks** from the nodes to the Internet and to other nodes (gray).

Discovery: The first step in composing a pervasive service is for the orchestrating mobile node to discover the nearby microcontroller devices and their capabilities. Since we assume an unstructured and non-fully-connected system, we cannot rely on a centralized discovery service and instead need a distributed discovery function. This can be achieved through broadcast beaconing based discovery services, such as Bluetooth Low Energy (BLE). Each of the platform instances encodes beacons with its capabilities as an execution platform (available sensors, actuators, computing), current status (load factors, execution queue lengths), policies (admission control, QoS), contact details (Wi-Fi SSID) and other metadata, and continuously broadcasts them around.

The mobile orchestrators will continuously scan the environment for the beacons, which allows them to maintain a view of the capabilities of the surrounding microcontroller devices. Based on this information, the orchestration logic will continuously decide which functionality (e.g., which scripts) should be installed in which devices in order to compose the desired services for the user. The complete service is developed as a set of scripts, along with the logic that describes how those scripts are to be deployed in the environment—the details are outside the scope of this paper.

Contacts: Once the orchestrating mobile has discovered the surrounding nodes and decided which scripts should run in which nodes, it needs to install the scripts. This is done via one-to-one contacts between the mobile and the microcontrollers. Any communication technology can be used for these contacts, but in our design and implementation we use Wi-Fi. In particular, the microcontroller devices all create their own Wi-Fi access points, whose SSIDs are advertised via the discovery function. This allows the mobile device to connect to the microcontrollers as a Wi-Fi station with a TCP/IP protocol stack for communications, and a custom protocol for transferring the scripts and their metadata from the mobile to the executor node—security mechanisms can be applied at various layers of the protocol stack. After installing a script in a device, the mobile disconnects and installs another script in another device, until the full service is created. The discovery-contact process runs continuously as the mobile device moves around and attempts to orchestrate newly discovered microcontrollers to run its services.

Backhaul and sidelinks: Aside from the communications used for discovery and contacts, it is also possible that some nodes are connected to the Internet via a backhaul network, or to each other via side links. In our initial design we do not explicitly consider these capabilities for the use in the platform functionality (e.g., for installing scripts over the Internet), but the runtime system may expose these resources to scripts via the Networking APIs.

2.3 Security Considerations

Our system is designed to execute code received from external sources, which raises significant security concerns. The primary threats are: 1) code execution by unauthorized users, 2) malicious code execution by authorized users, and 3) denial-of-service (DoS) attacks by, e.g., flooding the system with scripts. We divide security considerations into *admission*

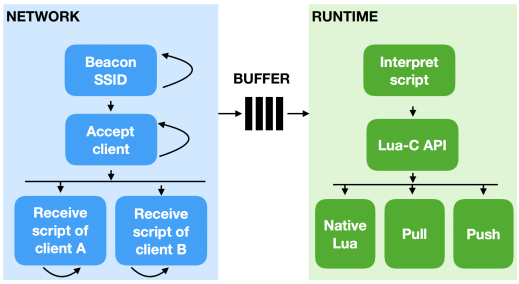

Figure 3: Execution environment for Lua scripts.

control (accepting or rejecting scripts), and runtime *execution control* (limiting what the scripts can to do). We control admission by the *source* and the *content* of the script. To control the source of the script, we need to authenticate the source credentials, e.g. by applying Wi-Fi security mechanisms. Potentially dangerous content could be detected via static analysis and by heuristics. During runtime, we control the executing code with an interpreting virtual machine, like the Lua VM.

3 PROOF-OF-CONCEPT

Our networked Lua platform offers dynamic services. For this, the platform consists of a network component, a runtime component, and a *buffer* connecting them. See Figure 3:

The **network** component receives Lua scripts from clients and sends platform responses to clients. It pushes received scripts into the **script buffer**. Concurrently, the runtime module pops the next script from the buffer, interprets it, replies via the network to the client of the script, and then pops the next script. The **runtime** component interprets Lua scripts, and accesses sensors and actuators. See Fig. 1b.

3.1 Lua Scripts

A Lua script is a sequence of Lua instructions. A Lua instruction is either a native Lua call or a Lua hardware call: We define a **native Lua call** directly in Lua and do not use any library. Examples for native Lua calls, therefore, include print statements, the return of a constant, and the addition. A **Lua hardware call**, in contrast, uses a C library: Lua embeds C in itself by linking Lua functions to C functions defined in a C library. Our dynamic execution environment accordingly implements access to its hardware resources in C. A Lua hardware function then points to its corresponding C function. This is used to implement the hardware drivers and other functionality that needs support from the OS.

3.2 Implementing the Runtime

The runtime component executes Lua scripts. To do this, the **runtime** first pops the next Lua script from the script buffer. To run this script, it passes it to the Lua-C API.

The **Lua-C API** interprets the Lua script. For this, the Lua-C API takes Lua commands, executes them, and returns calculation results and a status code. The Lua-C API—depending on the type of Lua script—executes a script either in a native, a pull or a push environment. The Lua-C API executes native Lua calls in the **native Lua environment**, one-time hardware calls in the **pull environment** and repeated hardware calls in the **push environment**. For example, the API executes a print statement in the native Lua environment. If the dynamic Lua platform needs to measure the temperature, map the measured temperature value to the color of an RGB LED, and finally turn on the RGB LED in the appropriate color, this is a one-time hardware call in the pull environment. If the dynamic Lua platform is to beacon the air pressure, this is a repeated hardware call in the push environment: The client sends our platform a beacon advertisement call that references a pressure read function. The push environment calls it periodically to measure and beacon the pressure.

3.3 Implementing the Networking

For enabling contacts from mobile nodes, we configure the Lua framework to act as a Wi-Fi access point. The discovery function is implemented via BLE, which beacons out the capabilities and the SSID of access point. From all the access points, a client then selects the one that has the required resources. The platform then accepts connection requests from clients and can receive their scripts in parallel.

4 EVALUATION

We evaluate the proof-of-concept prototype to provide initial results on the performance of our design. We present the evaluation in two parts: 1) micro-level tests that focus on the performance characteristics of a single platform instance, and 2) macro-level tests that focus on the performance of orchestrating multiple platform instances.

The experiments are carried out in a testbed with ten ESP32 microcontrollers with RGB LEDs as actuators and BMP180 as temperature/pressure sensors. The ESP32s run at their highest CPU frequency of 240 MHz and use the ESP-IDF v4.0-dev-837 distribution. The tests were carried out during the day in an office environment with multiple external Wi-Fi networks and BLE devices causing interference. A MacBook Pro laptop running MacOS 10.14 was used as the source of the Lua scripts (micro tests) and the orchestrator (macro tests), with all measurement scripts executed with Python 2.7. All code are publicly available (https://tinyurl.com/yxm3c3gy).

4.1 Micro Tests

The first thing we want to understand are the performance limits of dynamically loading scripts for execution. To this end, we connect a laptop to an ESP32 device and send it the

smallest possible script to interpret 100 times ("return 'OK';"). The mean end-to-end delay measured on the laptop from writing the script to the TCP socket to receiving a response from the socket is 8.35 ± 5.65 ms (median 6.69 ms). In contrast, the normal way of loading code into the same microcontroller (flashing a minimal binary) takes a mean of 46 ± 1 seconds—four orders of magnitude longer.

To separate the networking delays from the execution delays, we measure the timings on the ESP32. The delay on the microcontroller from just after receiving a buffer from the network to just before writing the response to the socket is 2.40 ± 0.27 ms, which means the communication delays are 71% of the total. The delay on the ESP32 from passing the script to the Lua VM to receiving the response is 1558 ± 199 μs, i.e., the total end-to-end delay breaks down to roughly 20% execution and 80% various overheads. *I.e., An efficient system must amortize overheads of larger execution tasks and pipeline executions of multiple scripts from a single client.*

As explained in §2.1, the typical scripts will interact with the device hardware by calling from Lua to the C driver to the OS. To understand the cost of these calls, we create two test cases; one where the Lua script sets the RGB LED color (ends up as calls that change the state of the ESP32's pulse width modulation, PWM, hardware), and one that reads the BMP180 sensor value (ends up as calls that communicate over the I2C bus). For the RGB LED, the total end-to-end execution time from the client to the ESP32 and back is 10.4 ± 4.2 ms (2 ms more than before). Out of this, the total VM execution time is 4170 ± 600 μs (40% of response time) out of which the time spent in the C function that configured the PWM hardware is 468 ± 66 μs. *I.e., the more complex Lua script causes the execution time to increase, but even accounting for all the overheads the platform should still be able to serve over a hundred concurrent clients.* For the BMP180, we measure 44.0±20.6 ms (end-to-end), 34.6±1.4 ms (VM execution), and 31.2±1.2 ms (sensor)—only allowing for about ten concurrent clients after overheads. This shows the significant time it takes to interact with some sensors, and suggests that access to the sensor data should be managed by the framework rather than having scripts directly manipulate the sensors.

Finally, we want to understand the behavior of the system under load. In principle, if scripts come in faster than the platform can execute them, a queue should build up between the network and runtime components (middle of Figure 3), which should show up as increased end-to-end delays. To set up an overload scenario, we start n concurrent clients, each repeatedly sending Lua scripts (total 20 each) that busy-loop for 100 ms. We observe the mean delay grow according to $D = 80n + 42$ (linear regression, $R^2 = 0.9967$), where D is the end-to-end delay for a single script execution in milliseconds—for $n = 1$ the mean is 111 ms and $n = 9$ it is 750 ms, and with $n = 10$ the ESP32 locks up and the test does not finish. 90th percentile delay grows according to $D = 106n + 8$ ($R^2 = 0.9978$). These show that the system behaves as expected under load (each additional client means roughly 100 ms more waiting time), but locks up when the offered load exceeds the processing capacity.

4.2 Macro Tests

Finally, we want to understand the basic performance characteristics of orchestrating multiple framework instances into a simple distributed system. To do this, we set up a testbed with ten ESP32 devices running the Lua platform and used a laptop to orchestrate them. The Lua devices each create a Wi-Fi access point (AP) and advertise its SSID, along with its capabilities (e.g., which sensors it has), via BLE beacons. The orchestration process has two stages: First, the orchestrator scans for BLE beacons to discover the nearby nodes and their capabilities. Second, the orchestrator connects to the Wi-Fi AP of each device, one after another, and installs a Lua script. When all scripts are installed, the process is complete.

For the discovery phase, we use the most aggressive advertising settings available in the ESP32 device, with an advertising interval of 20–40 ms. We measure the total discovery phase (i.e., discovering all ten devices) to take a mean of 224 ± 81 ms and 331 ms 95th percentile delay. The mean time to discover an individual device is 75 ± 80 ms, with a median of 33 ms and 232 ms 95th percentile. *I.e., the discovery process is near instantaneous—but will increase if the beaconing is less aggressive or there are more devices.*

The installation phase is run on Mac OS 10.14 as a Python 2.7 script that uses the `''wireless''` library to associate with the ESP32 Wi-Fi APs (802.11n, 2.4 GHz, automatic channel selection, WPA2 PSK security), connect to the TCP server running on the device, and then install a Lua script—repeated until all ten devices are set up. The mean delay to orchestrate all ten devices is 99.4 ± 2.6 s, with an individual device taking a mean of 9.9 ± 1.1 s to configure. Out of this time, the 97% goes into associating with the AP (9.6 ± 1.0 s for a single device), after which connecting the TCP socket and installing the script takes only a few hundred milliseconds. *The association time could likely be reduced by using more optimal Wi-Fi control methods, but is still likely to remain the limiting factor when orchestrating devices— new technologies like Wi-Fi Aware may help.*

5 RELATED WORK

In general, our work fits into *edge-oriented computing* [8] that has too many publications [9, 11] to cover here. Instead, we relate three characteristics of our work to previous work.

First, we do not assume any kind of structured infrastructure supporting our system, but rather rely only on direct

opportunistic manipulation of nearby devices. This is in contrast to much of the edge-computing work, which considers well-structured, integrated systems composed of a cloud, edge-gateways and device elements [14]. For example, typical edge-supported IoT architectures would be composed of a centralized cloud controller, connected to a fog-layer with gateways serving the "things", with raw data and data processing being pushed towards the edge and the intelligence centralized at the cloud [2]. *Mist computing* [3] takes this a step further by pushing computational tasks all the way to the microcontrollers at the edge—this is what we propose also, but with direct opportunistic orchestration rather than being a part of a multi-tiered cloud-edge-fog architecture.

Second, our goal is not offloading—which mainly attempts to get around the limited computational and energy resources of mobile devices [10]—but rather the dynamic programming and orchestration of heterogeneous devices to change the behavior of the users' environment. While the mobile edge offloading systems aim to bring general compute resources close to the user—e.g., through well-connected *cloudlets* [12] or by exploiting disconnected infrastructure via delay tolerant mechanisms[1]—to enable constrained mobile devices offload heavy computations, we instead focus on special purpose appliances like clothes irons and integrated smart infrastructure like air conditioning controllers.

Finally, our focus is not on short-term tasks that run a computation and return a result, but rather in continuously running logic, such as control loops. The idea of executing small stateless scripts in response to network events has been productized in the past few years as *serverless computing* [6] by multiple cloud vendors (e.g., AWS Lambda and Google Cloud Functions). This idea has recently been extended for *serverless computational offloading at the edge* [4], with well-structured system of dedicated lambda executors and dispatchers that intelligently choose where lambdas offloaded from user devices are to be executed. While the basic idea of moving a stateless script from an end-user device to a nearby node for execution is the same in our work, the rest of the system is different both in architecture and in vision.

There is previous work that shares our goal of making the environment around us programmable. One early example of this is the *Gator Tech Smart House* [5], which includes a platform for dynamically programming smart home sensors and actuators. In contrast to our approach of directly loading code into the sensor/actuator microcontrollers, their sensors execute fixed firmware that communicates with a gateway that executes "surrogate" code that abstracts the device into a service available for other code. The authors of [13] share our vision of a *programmable world* where developers can write code to effortlessly run on the IoT devices around us.

6 CONCLUSION

In this paper we have taken a step towards a *programmable world* by demonstrating the feasibility of turning microcontrollers into execution environments for custom scripts. We showed that the cheap, commercially available ESP32 microcontrollers can execute scripts for at least tens of nearby users, allowing it and its attached sensors and actuators to be dynamically programmed by nearby users. Further, we demonstrated that multiple microcontrollers can be orchestrated to cooperatively generate new hyperlocal services that are deeply integrated to the physical environment.

REFERENCES

[1] Mohammad Aazam, Khaled A Harras, and Ali E Elgazar. 2018. Delay Tolerant Computing: The Untapped Potential. In *MobiCom CHANTS*.

[2] Yuan Ai, Mugen Peng, and Kecheng Zhang. 2018. Edge computing technologies for Internet of Things: a primer. *Digital Communications and Networks* 4, 2 (2018), 77–86.

[3] Rabindra K Barik, Amaresh Chandra Dubey, Ankita Tripathi, T Pratik, Sapna Sasane, Rakesh K Lenka, Harishchandra Dubey, Kunal Mankodiya, and Vinay Kumar. 2018. Mist data: leveraging mist computing for secure and scalable architecture for smart and connected health. *Procedia Computer Science* 125 (2018), 647–653.

[4] Claudio Cicconetti, Marco Conti, and Andrea Passarella. 2019. Low-latency distributed computation offloading for pervasive environments. In *IEEE PerCom*.

[5] Sumi Helal, William Mann, Hicham El-Zabadani, Jeffrey King, Youssef Kaddoura, and Erwin Jansen. 2005. The gator tech smart house: A programmable pervasive space. *Computer* 3 (2005), 50–60.

[6] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Serverless computation with openlambda. In *USENIX HotCloud*.

[7] IC Insights, Inc. 2018. The McClean Report.

[8] Chao Li, Yushu Xue, Jing Wang, Weigong Zhang, and Tao Li. 2018. Edge-oriented computing paradigms: A survey on architecture design and system management. *ACM Computing Surveys (CSUR)* 51, 2 (2018), 39.

[9] Hang Liu, Fahima Eldarrat, Hanen Alqahtani, Alex Reznik, Xavier De Foy, and Yanyong Zhang. 2017. Mobile edge cloud system: Architectures, challenges, and approaches. *IEEE Systems Journal* 12, 3 (2017), 2495–2508.

[10] Pavel Mach and Zdenek Becvar. 2017. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials* 19, 3 (2017), 1628–1656.

[11] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B Letaief. 2017. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys & Tutorials* 19, 4 (2017), 2322–2358.

[12] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing* 4 (2009), 14–23.

[13] Antero Taivalsaari and Tommi Mikkonen. 2017. A roadmap to the programmable world: software challenges in the IoT era. *IEEE Software* 34, 1 (2017), 72–80.

[14] Antero Taivalsaari and Tommi Mikkonen. 2018. A Taxonomy of IoT Client Architectures. *IEEE Software* 35, 3 (2018), 83–88.