Devices-as-Services: Rethinking Scalable Service Architectures for the Internet of Things

Fatih Bakir, Rich Wolski, Chandra Krintz
Univ. of California, Santa Barbara

Gowri Sankar Ramachandran
Univ. of Southern California

Abstract

We investigate a new distributed services model and architecture for Internet of Things (IoT) applications. In particular, we observe that devices at the edge of the network, although resource constrained, are increasingly capable – performing actions (e.g. data analytics, decision support, actuation, control, etc.) in addition to event telemetry. Thus, such devices are better modeled as servers, which applications in the cloud compose for their functionality. We investigate the implications of this "flipped" IoT client-server model, for server discovery, authentication, and resource use. We find that by combining capability-based security with an edge-aware registry, this model can achieve fast response and energy efficiency.

1 Introduction

As the Internet of Things (IoT) grows in size and ubiquity, it is becoming critical that we perform data-driven operations (i.e. analytics, actuation, and control) at the "edge" to reduce the latency, response time, cost, and energy use for IoT applications. As such, edge systems increasingly co-locate data management and analysis services with sensing, instead of requiring that devices ship their data over long-haul networks for remote processing using the traditional "cloud" model.

Edge systems [3, 4, 8, 10, 11, 27, 28] typically implement a publish/ subscribe (pub-sub) model in which devices publish streams of data (often via a nearby broker); when supported, actuation often uses a separate protocol. Alternatively, some solutions target a client-server model, where the IoT devices are the clients that respond with data whenever a decision making server needs it. In our view, a client-server model is well suited to emerging IoT applications in which resource constrained edge devices provide *nanoservices* – data-driven actuation and control, data analysis, processing, *and* sensing.

We believe that the IoT deployments in the not-too-distant future will include multi-function devices. As a result, a services model in which the specific function (including data publication) can be requested from each device when it is needed is more appropriate. Further, because devices will continue to be resource constrained, they will require "helper" services at the edge that augment device capabilities and enable scale. In this paper, we outline this approach to implementing *Devices-as-Services* and describe some of the capabilities of an early prototype.

Our work is motivated by the following observations.

- IoT applications can and will likely be structured as collections of services that require functionality from a device tier, an edge tier, and a cloud tier
- in-network data processing can significantly reduce response time and energy consumption [31],
- edge isolation precludes the need for dedicated communication channels between application and devices, and facilitates privacy protection for both data and devices,
- actuation in device tier will require some form of request-response protocol where the device fields the request,
- the heterogeneity of devices militates for a single programming paradigm and distributed interaction model, and
- multi-function devices can and will be able to perform their functions (including data publication) conditionally based on application needs (e.g. as a power optimization).

Thus we propose a new model for distributed IoT applications in Figure 1. We *flip* the client-server model such that devices at the edge are "servers" that although resource constrained, service multiple, scalable applications (i.e. clients) deployed in the cloud. Note that in many commercial solutions [3, 11] devices "publish" data to channels to which servers (running in the cloud) subscribe. We propose to move the servers to the edge (as a way to scale data ingress, lower latency, and improve privacy) thereby reversing the current cloud-centric architecture. Such a model requires a new distributed architecture that leverages edge resources in these ways.

In this paper, we define one such architecture, investigate how it can accommodate this new model, and evaluate the implications of its use in IoT settings and for servicing IoT applications. The architecture is based on the functions as-a-service (FaaS) programming and deployment model (the execution engine that underpins serverless computing for clouds [13, 20, 22, 23]), and integrates a new approach for efficient client and server discovery, authentication, and authorization via a combination of capability-based security [5, 12, 24]

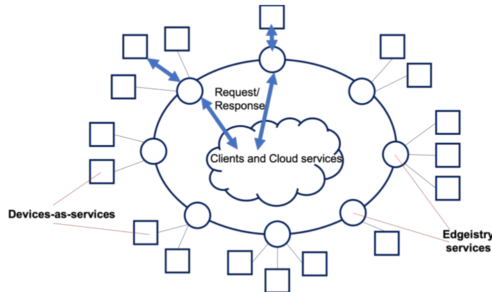

Figure 1: *A new distributed services model for IoT: "client" applications in the cloud compose services exported by resource-constrained devices at the edge via Edgistries – edge nodes that facilitate device discovery/registration, privacy mediation, and optimization.*

and a set of edge services for service registry, privacy mediation, and optimization called an *Edgistry*.

We prototype an early implementation of this architecture and approach using microcontrollers, single board computers, and edge clouds. We empirically compare the energy consumption and performance of traditional and flipped architectures, as well as our capability approach versus TLS/SSL based mechanisms for authentication. We find that it is possible to implement Devices-as-Services – a *flipped* model of the currently popular cloud-based IoT architecture – efficiently, using FaaS as a universal programming paradigm and a set of application agnostic edge services.

2 Devices-as-services – a New Approach

Our view is that the current Internet and cloud architecture should "reverse" to accommodate scalable IoT applications. Rather than hosting services in the cloud (logically making IoT devices clients of those services), we view devices as "servers" and applications running in the cloud as "clients".

This viewpoint is supported by several observations. First, devices are resource restricted making them capable of delivering limited and relatively fixed functionality. This functionality is naturally described as an enumerable set of services (responses to requests) that are composed by applications. These services are necessarily "small" but even fully resourced cloud services are now being developed as microservices [18]. Devices are the "nanoservices" in an IoT setting.

Secondly, applications must compose device capabilities from vast collections of devices into meaningful functionality for users, and in an IoT setting, the user scale will be substantial. The cloud is where this scaling will necessarily take place, both in terms of aggregating device functionality, and matching these aggregations to user demand for them.

Thirdly, much of the current technological development for IoT [3, 11] is focused on bringing devices to the cloud. Many of the commercial offerings provide an SDK for using MQTT [4, 28] or AMQP [1] to publish telemetry, events, etc. to objects (which subscribe to device channels) in the cloud that represents each device. For actuation, each device must separately subscribe to such a channel to receive asynchronous commands (although for low-power applications that use MQTT-SN, this option is not possible). With our system, the same server code runs at all tiers — device, edge, and cloud — unifying the device API as a first-class services API.

The currently prevalent commercial IoT/cloud APIs require devices to act separately as "publishers" or "subscribers" or both (the last to implement "closed-loop" actuation). We believe that IoT infrastructure must accommodate a richer model in which devices (however resource restricted) are capable of actions as well as event telemetry. Thus, logically, IoT devices are better modeled as servers that provide services to applications.

2.1 The Edgistry

Our approach splits the provisioning of services between devices and a common set of management services located at the edge, termed *The Edgistry*. Devices host small, resource-light services that field requests from, and respond to, client applications (hosted in the cloud or edge). The Edgistry

- implements an eventually consistent distributed service registry (e.g. using blockchain consensus protocols such as Tendermint BFT [6] and Hyperledger Fabric [2]),
- acts as a speed-matching communication service,
- can protect device privacy (e.g. through anonymization) by isolating the service provider from service consumer, and
- provides computational and storage off-loading services (e.g. content caching) for device-hosted services.

From a programmability perspective, we advocate a universally portable "Functions as a Service" (FaaS) capability [13, 17, 20, 22, 23]. Services on the device are implemented using a "micro FaaS" – a small, specially tuned, implementation of FaaS specifically targeting resource-restricted microcontrollers as an execution platform. The micro FaaS supports the same programming APIs as a heavier-weight (and serverless) implementation designed to run on an edge device, in a private cloud, or in a public cloud. In this way, IoT applications can be coded using a single "FaaS everywhere" set of programming abstractions from device to cloud.

To enable this portability and also data durability in a distributed setting, such scale-spanning FaaS capability must define a portable storage abstraction, ideally having append-only semantics. Thus all computations, regardless of location, can persist data by appending it to some object hosted in the device-edge-cloud hierarchy using a common API.

2.2 Request Forwarding and Duty Cycle

Note that to save power, some devices will need to spend most of their duty cycle in a power-saving "sleep" mode. For example, an ESP8266 microcontroller [9] 0.01 mAh (milliamp-hours) in deep sleep mode and requires 320 mAh to transmit a packet using its on-board WiFi. Using a 18650 Lithium-ion rechargable battery, it is possible to operate the microcontroller for approximately 1 year without a battery recharge if it limits its communication to every 10 minutes (on the

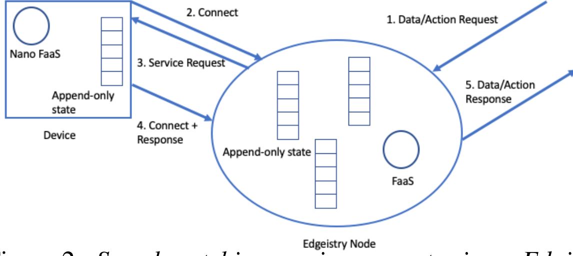

Figure 2: *Speed-matching service requests via an Edgistry*

average, using WiFi and WPA2) [29]. Thus, the network connectivity we can expect will be initiated by the device on a duty cycle dictated by its power budget.

Note that in a "typical" server setting, the client initiates a connection to the server when making a request. Thus the server's activation can be triggered by the connection initiation. In this setting, the connection and the request-response interaction are decoupled. That is, the device (running a service) polls the Edgistry to see if in-bound requests are queued there. Any pending requests will be forwarded to the device which can then disconnect (to save power). When a response is ready, the device connects to the Edgistry and sends a response. Moreover, both the service running on the device and the proxy running on the Edgistry uses append-only data structures to store state waiting for successful transmission.

Figure 2 shows how a request for service from a device (registered in a serachable registry, not shown) arrives at the Edgistry and is appended to a request queue associated with the device. When the device contacts the Edgistry (as part of its duty cycle), the request is forwarded to the device and appended to a queue of requests, thereby triggering a function in the micro FaaS. When the response is ready, the device connects and sends it to the Edgistry where it is appended to a queue of responses destined for a client. Finally, the Edgistry forwards the response to the original requester.

2.3 Device Registry and Privacy

The Edgistry must implement service discovery (i.e. for devices implementing services) and client registry. That is, when a client makes a (possibly speed-matched) request from a service on a device, the Edgistry will need to keep track of where the response must be returned. We envision each device pairing with a one local Edgistry node (or a small number of local nodes) so that it does not have to store per-request client tracking information.

Service discovery can be implemented using a distributed, eventually consistent blockchain consensus protocols such as Tendermint BFTT [6]. A blockchain consensus model is particularly attractive because it offers the opportunity for the Edgistry to implement privacy features such as location and identity anonymization [21]. For example, an application may wish to contact a device within some geographic region without the need to know the precise location of the device. This type of location "fuzzing" can be implemented using policy delegation by the device to the Edgistry node with which it is paired. By hiding the identity of the device from the service requesters, we can mitigate denial of service attacks as the Edgistry delegates the service to one of the devices in the network in a geographical neighborhood without revealing the device identity to service consumers.

3 Capability-based Service Access

Security is a challenge for this inverted model because it requires distributed policy implementation governing a vast number of services, each implemented using the barest minimum of computational and storage resources. This latter requirement alone precludes the use extant public-key/private-key "standards" for securing client-server interactions. Specifically, asymmetric cryptography computations and key storage are costly for resource restricted devices. Moreover, current TLS [26] protocols do not allow servers to control packet size, so servers must have large memories (e.g. 16k for half duplex, 32k for full duplex) to conform.

Our approach uses a distributed, capability based authentication scheme to address these challenges. A capability is a communicable, unforgeable token of authentication. It encapsulates an object, the access rights on that object, and a cryptographic signature that maintains the integrity of the capability. We implement access rights as a bitmap and generate a signature using a private key based method such as Hashed Message Authentication Codes (HMACs) [19]. The signature protects the object name and access rights in the capability. To verify the token, the server receiving a request regenerates the signature from the capability body and compares it against the capability signature. If it matches, the server executes the request and discards the capability.

3.1 Controlled sharing without the server

Capabilities can support privilege reductions without involving the server. For example, Amoeba [24] implements a derivation mechanism that uses commutative hash functions on access-right bitmaps to selectively reduce capability rights.

We generalize this derivation mechanism to support a wider variety of policy implementations. To enable this generalization, each capability maintains a derivation history. Using this history, the server can verify the validity of derivations. On each derivation, the signature of the current chain is merged with the hash of the new capability. Upon receiving a request, the server walks the chain applying the hashes. If the signatures match at the end and each derivation is legal (e.g. does not add new rights), the derivation is deemed valid, and the request is served. This mechanism is similar to Macaroons [5] for web services, but with optimizations that permit multiple entries to be combined in a single signature generation. Because any holder of a capability can extend the chain of rights constraints this capability system can implement truly distributed access control.

It is possible for our derivation chains to grow over time. To bound this growth we provide a flattening functionality, which compresses a chain of derivations into a single capability upon each successful request. Because server verification of

capabilities is efficient (cf. Section 5) this methodology is appropriate for very large scale IoT deployment.

3.2 Protecting Capabilities

To avoid the use of resource-consuming encrypted links between devices at the edge, we devise a novel approach to protecting capabilities in clear text channels. In our scheme, capabilities are further constrained to a specific request before being transmitted. Thus, a capability transmitted over the network can only be used for that request. We number requests with monotonously increasing sequence numbers to prevent replay attacks. To enable this, we employ the current time as the sequence numbers of each request. Since updating the sequence number in the server requires a valid capability, a DOS attack cannot be employed by an outside attacker. The device stores the last sequence number for efficiency purposes. If a client has a buggy time protocol implementation or a clock drifts, our protocol handles this case by returning the current sequence number in the error response. The clients can synchronize their clocks using this number. Upon the detection of such a drift, an external service such as the Edgistry can fix the sequence number using a special capability. If the attack model of the application involves malicious clients, a separate sequence number can be employed per client.

3.3 Maintaining trust

The end device and Edgistry perform 2-way authentication so that either can verify requests from the other. During bootstrap, the end device passes a capability to the Edgistry. Using 2-way authentication, the Edgistry also passes a capability it signed to the end device. Using this capability, the Edgistry can verify the authenticity by requiring the end device to send responses by deriving from that capability.

4 A Prototype

We prototype this "flipped" client-server model and our capability-based security method using CSPOT [30], an open source [1] distributed platform for deployment and execution of IoT applications. CSPOT runs as a FaaS server on microcontrollers (i.e. as a micro FaaS), and as a serverless FaaS runtime system on edge devices, private clouds, and public clouds. In CSPOT, a function invocation *can only* be coupled with a "Put" of data targeting some storage object, which is append-only and persistent. The function coupled with the put has direct access to this newly posted data as well as all previously appended data up to some limit specified with the object is created. CSPOT data objects are called WooFs (**W**ide-area **o**bjects **o**f **F**unctions) and each WooF resides in a single namespace. Namespaces are addressed by a URI and, thus, network accessible. A CSPOT application can only persist data in WooFs located in one or more namespaces – the functions that are invoked are thus stateless while executing.

In our prototype implementation, we use capabilities to authenticate four CSPOT functions: namespace creation, WooF creation within a namespace, Put operations which store data in a WooF (and may invoke a function), and Get operations which retrieve data from a WooF.

Upon flashing a device, we generate an HMAC secret. A "root" capability is securely issued when the device is paired with an Edgistry node. This capability carries the right to create a namespace on the device which the Edgistry node uses to initialize the service.

As part of the bootstrapping process, the Edgistry creates the necessary namespaces and the WooFs on the device. After the CSPOT components are initialized, the application, for instance a temperature monitor, is started and periodically Puts a new reading into the local WooF.

Note that because the Edgistry has the initial root capability, it can implement all access control policies via derivations from the root capability once the initial trust relationship is established. For instance, the Edgistry can generate capabilities for the WooFs and namespaces it creates during the bootstrap process, rather than the device generating and transmitting them. The device need only verify each derivation, thereby saving code complexity and power.

At the end of the bootstrapping process, the Edgistry derives the application capabilities. For instance, the Edgistry has full access rights to the WooFs in the device, but it derives and transfers a read only capability for the clients (applications) to use, following the principle of least privilege.

Policy delegations also can be performed by clients. For instance, an application client A can share a capability C_1 with another application B with a constraint that the derived capability C_1' is only valid in the presence of another capability, C_2, that B must present alongside C_1'. Thus clients can also create derivations that implement policies such as identities (represented by an identity capability) without the server's involvement (i.e. without contacting the device which can always verify *any* derivation).

Table 1: Comparison of Devices-as-services and AWS IoT. Units are milliseconds; across 100 benchmark runs.

	mean	stdev	max
AWS IoT+λ	5578	265	6843
CSPOT device->Edgistry->AWS	608	5.78	652

5 Evaluation

Our initial goal is to verify the execution and power efficiencies of this new model at the device level. We begin with an abbreviated comparison of the Devices-as-services model to the IoT infrastructure offered by Amazon AWS [16]. Table 1 shows the end-to-end timings from a ESP8266 [9] microcontroller to AWS using our new model, and the AWS IoT SDK coupled with AWS Lambda.

This benchmark uses NTP-synchronized clocks to record a timestamp on the device and another in AWS; it computes end-to-end latency as the difference between the two. Both CSPOT and AWS Lambda implement an event-driven FaaS programming environment. For AWS, we persist in DynamoDB [7]

[1] https://github.com/MAYHEM-Lab/cspot.git

For the CSPOT case, we replicate data on an Edgistry node (an Intel NUC) [15] in a WooF before forwarding/persisting it in CSPOT running in a virtual machine in AWS. The table shows mean, standard deviation, and maximum latencies in milliseconds over 100 experimental runs. The Edgistry node is hosted using an edge cloud [8] running Eucalyptus [25] located in the same room as the microcontroller. The edge cloud is connected to the UCSB campus network. From this data (and a more detailed comparison that includes Azure IoT Hub [30])it is clear that this new methodology is at least an order of magnitude faster (in terms of latency) than comparable commercial offerings even when it replicates data in the Edgistry. Further, the coefficient of variation for the AWS experiment is 0.04 and 0.009 for the CSPOT experiment, indicating that our system is also an order of magnitude more stable in terms of performance variation.

Moreover, devices that publish all of their data all of the time (e.g. using MQTT to leverage AWS or Azure) wastes precious battery lifetime when that data is not demanded. This new model allows devices to function as servers and only to respond when data is requested by a client application. Thus, in this work, we focus on improving the efficiencies of the security features at the edge, where computational power and electrical energy (i.e. battery power) are at a premium.

Much of the latency experienced in commercial offerings (and the concomitant loss of battery life through additional active time in the device) is associated with the TLS-based security protocols that MQTT and AMQP implementations use. While it is possible to use the bidirectional nature of MQTT and AMQP communications to implement close-loop device interactions, these protocols (let alone the APIs that actuate them) are not specifically designed to implement higher-level service interactions on devices with moderate or severe power restrictions and/or very limited memory. Devices-as-services requires a more efficient, high-level interaction and key to that interaction is efficient authentication. We next compare our capability-based approach (which uses an HMAC-based mechanism) to competitive approaches. In all experiments, SHA256 hash was used for generating a digest from messages. For the public key cryptography experiments, we use the recommended key size for RSA of 2048 bits and 4096 bits and 256 bits for ECC. For the hash generation, we use a custom version of libemsha. Generation of SHA256 hashes from messages are common to both approaches. Our implementation takes around 88 (0.06) microseconds per SHA256 hash on a message of 32 bytes. The time it takes to hash a message scales linearly with message size.

Table 2 shows a comparison of the execution times for various cryptographic techniques when used to sign and verify messages. The table shows the times for RSA (using PKCS1) for two different key lengths, ECDSA (an Elliptic Curve Cryptography – ECC – method popular in many IoT applications [14]), and an HMAC-based scheme we have implemented. Below the double lines we also show the performance of a 3-level capability derivation and a 5-level derivation on the server (e.g. to represent a setting in which the clients constrain capabilities multiple times). We also ran a TLS based server experiment. We find that each connection uses 3950 milliseconds (ms; standard deviation (stdev) 11 ms) and 32320 ms (stdev 5 ms) for a simple TCP request and 2048 bit and 4096 bit keys, respectively. The difference between these two measurements is due to network performance variability.

Table 2: Comparison of cryptographic algorithms for signing and verifying a 32 byte message on the ESP8266. Average execution time and standard deviation (in parens) are shown. Last 2 rows show end-to-end measurements.

Algorithm	Sign ms (stdev)	Verify ms (stdev)
PKCS1 (2048 bit)	3280 (190)	187 (4)
PKCS1 (4096 bit)	31580 (190)	9190 (9)
ECDSA (256 bit)	214 (1)	4340 (216)
HMAC (64 bit)	0.37 (0)	0.37 (0)
HMAC (128 bit)	0.37 (0)	0.37 (0)
3-level Derived Capability (64 bit)	0.77 (0)	1 (0)
5-level Derived Capability (64 bit)	1.18 (0.04)	1.3 (0)

Clearly, an HMAC-based approach is considerably less computationally intensive than either of the competitive approaches. Indeed, even the derived capability timings are better for a 3-level derivation than for a single capability verification using either of the other schemes.

While the results in Table 1 indicate that, overall (even with additional persistent data replication) our method requires an order of magnitude less "active time," and active time is proportional to power usage, we highlight on the energy efficiency of the authentication protocol. Specifically, our method uses 0.072 mJ and 0.185 mJ for generation and verification, respectively. RSA signatures (2048 bit keys) consume 606.1 mJ and 34.6 mJ, and ECC signatures use 39.54 mJ and 80.32 mJ, respectively. That is for capability generation our method uses between 3 and 4 orders of magnitude less energy for authentication than either RSA or ECC.

6 Conclusions

We propose a new "flipped" client-server model for IoT in which devices at the edge are servers that provide nanoservices, which applications in the cloud (the clients) compose for their implementations. We contribute a novel approach to distributed service design based on "FaaS everywhere," edge-level support, and a novel capability mechanism for distributed policy implementation, a fuller exposition of which is available from [30]. Our empirical evaluation shows that this approach is feasible and introduces very little overhead and power consumption.

This research is supported in part by NSF (CNS-1703560, OAC-1541215, CCF-1539586), ONR NEEC (N00174-16-C-0020), AWS Cloud Credits for Research, and the USC Viterbi Center for Cyber-Physical Systems and the Internet of Things.

References

[1] Amqp home page. https://www.amqp.org, 2019. [Online; accessed 2-May-2019].

[2] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Eurosys*, page 30, 2018.

[3] Azure Internet of Things. https://www.microsoft.com/en-us/cloud-platform/internet-of-things-azure-iot-suite. [Online; accessed 22-Aug-2016].

[4] A. Banks and R. Gupta. Mqtt v3.1.1 protocol specification, 2014.

[5] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Network and Distributed System Security Symposium*, 2014.

[6] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, University of Guelph, 2016.

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Symposium on Operating System Principles*, 2007.

[8] Andy Rosales Elias, Nevena Golubovic, Chandra Krintz, and Rich Wolski. Wheres the bear?–automating wildlife image processing using iot and edge cloud systems. In *ACM Conference on IoT Design and Implementation*, 2017.

[9] ESP8266 Specifications Web Site, 2019. [Online; accessed 15-March-2019] https://www.espressif.com/en/products/hardware/esp8266ex/overview.

[10] Fog Data Services - Cisco. http://www.cisco.com/c/en/us/products/cloud-systems-management/fog-data-services/index.html. [Online; accessed 22-Aug-2016].

[11] GreenGrass and IoT Core - Amazon Web Services. https://aws.amazon.com/iot-core,greengrass/. [Online; accessed 2-Mar-2019].

[12] S. Gusmeroli, S. Piccione, and D. Rotondi. A Capability-based Security Approach to Manage Access Control in the Internet of Things. *Mathematical and Computer Modelling*, 58(5-6), September 2013.

[13] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Serverless computation with openlambda. In *HotCloud*, 2016.

[14] J Hernandez-Ramos, A. Jara, L. Marin, and A. Skarmeta Gomez. Dcapbac: Embedding authorization logic into smart things through ecc optimizations. *Int. J. Comput. Math.*, 93(2), February 2016.

[15] Intel NUC. https://en.wikipedia.org/wiki/Next_Unit_of_Computing [Online; accessed 1-Feb-2018].

[16] Internet of Things - Amazon Web Services. https://aws.amazon.com/iot/. [Online; accessed 22-Aug-2016].

[17] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. Popa, I. Stoica, and D. Patterson. Cloud Programming Simplified: A Berkeley View on Serverless Computing, Feb 2019.

[18] M. Jung, S. Mollering, P. Dalbhanjan, P. Chapman, and C. Kassan. Microservices on AWS. https://docs.aws.amazon.com/aws-technical-content/latest/microservices-on-aws/introduction.html, September 2017. [Online; accessed 2-Mar-2019].

[19] Hugo Krawczyk, Ran Canetti, and Mihir Bellare. Hmac: Keyed-hashing for message authentication, 1997. [Online; accessed 26-Apr-2019] https://tools.ietf.org/html/rfc2104.

[20] H. Lee, K. Satyam, and G. Fox. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, July 2018.

[21] L. Liu. Privacy and location anonymization in location-based services. *SIGSPATIAL Special*, 1(2), July 2009.

[22] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha. A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms. In *IEEE International Conference on Cloud Computing*, Dec 2017.

[23] G. McGrath and P. R. Brenner. Serverless computing: Design, implementation, and performance. In *International Conference on Distributed Computing Systems Workshops*, June 2017.

[24] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba – A distributed

Operating System for the 1990's. *IEEE Computer*, 23(5), May 1990.

[25] Daniel Nurmi, Richard Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *IEEE Cluster Computing and the Grid*, 2009.

[26] E. Rescorla. The transport layer security (tls) protocol version 1.3. RFC 8446, IETF, August 2018.

[27] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4), 2009.

[28] A. Stanford-Clark and H. Truong. Mqtt for sensor networks (mqtt-sn) protocol specification, 2013.

[29] Dr. Thorsten von Eicken's Low Power WiFi Blog, 2019. [Online; accessed 15-March-2019] https://blog.voneicken.com/projects/low-power-wifi-intro/.

[30] R. Wolski and C. Krintz. CSPOT: A Serverless Platform of Things. Technical Report 2018-01, UC Santa Barbara, 2018. https://www.cs.ucsb.edu/research/tech-reports/2018-01.

[31] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3), September 2002.