Poster: ECIoT: Case for an Edge-Centric IoT Gateway

Udhaya Kumar Dayalan, Rostand A. K. Fezeu, Nitin Varyani, Timothy J. Salo, Zhi-Li Zhang
Department of Computer Science & Engineering, University of Minnesota – Twin Cities, U.S.A.
{dayal007,fezeu001,varya001,salox049,zhang089}@umn.edu

ABSTRACT

Numerous cloud service providers (CSPs) have developed IoT gateways and devices that connect IoT devices solely to their respective clouds. We term these gateways *cloud-centric*. In this paper, we propose an alternative, *edge-centric* approach to developing IoT gateways, *Edge-Centric IoT Gateways,* or *ECIoT.* ECIoT gateways will: enable sites to direct IoT sensor data to the CSP of their choice for processing and storage and permit IoT sensor data to be easily and efficiently forwarded to multiple CSPs simultaneously, which can simplify these functions in multi-vendor IoT systems. Evaluation of a preliminary, proof-of-concept prototype suggests that the ECIoT gateway can achieve these multi-vendor objectives with minimal overhead.

KEYWORDS

IoT Edge Device, IoT Gateway, IoT Cloud

ACM Reference Format:
Udhaya Kumar Dayalan, Rostand A. K. Fezeu, Nitin Varyani, Timothy J. Salo, Zhi-Li Zhang. 2021. Poster: ECIoT: Case for an Edge-Centric IoT Gateway. In *The 22nd International Workshop on Mobile Computing Systems and Applications (HotMobile '21), February 24–26, 2021, Virtual, United Kingdom.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3446382.3448667

1 INTRODUCTION & MOTIVATION

Many cloud service providers (CSPs) have developed their own IoT gateways and devices. Typically, these gateways and devices connect solely to their respective cloud services. However, these vendor-specific IoT gateways often make it difficult to connect IoT devices to alternate CSPs. This "vendor lock-in" presents sites that have deployed IoT gateways and devices from multiple vendors with several challenges: 1) directing IoT sensor data to an alternative CSP, a CSP other than the one to which the gateway or device is "locked", for processing and storage is extremely difficult; 2) analyzing IoT data with a unified application that processes data generated by IoT devices supplied by multiple vendors becomes very challenging; at best, the data must be forwarded from multiple applications running on different CSPs to a common application for processing, rather than permitting the common application to receive the sensor data directly from the IoT devices, and 3) device management becomes cumbersome, because each vendor's IoT devices are locked to that vendor's device configuration and management tools.

HotMobile '21, February 24–26, 2021, Virtual, United Kingdom

ACM ISBN 978-1-4503-8323-3/21/02…$15.00
https://doi.org/10.1145/3446382.3448667

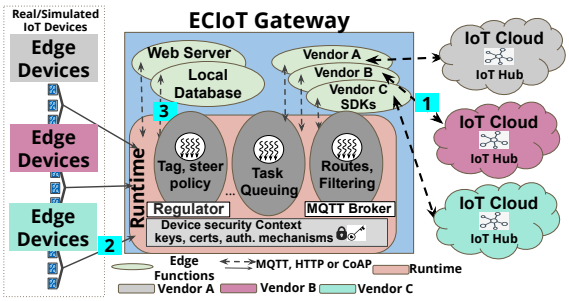

Figure 1: Edge-Centric IoT Gateway Framework

Several CSPs, including Microsoft [8], Amazon (AWS)[3] and Google [1]) have made available IoT gateway platforms, or software development kits (SDKs), that IoT vendors may integrate into their IoT devices or gateways. These CSP-provided SDKs simplify the development of IoT devices and gateways, but at the expense of locking the vendors into to the SDKs' respective cloud services. We term these commercial IoT gateways as *cloud-centric* , as their primary goal is to connect IoT devices and marshal IoT data to the CSP's cloud. Such cloud-centric solutions not only lead to "walled" IoT ecosystems that are not interoperable [5], but also create various management issues and performance inefficiencies.

In this paper, we advocate an *edge-centric* architecture for designing IoT gateways. Instead of merely connecting IoT devices to cloud services, we envision an *edge-centric* IoT gateway that i) leverages computing and storage capabilities at the network for edge-based device management and ii) exploit availability of multiple cloud services (from different vendors) for "best" (*E.g.,* fastest or cheapest) IoT data analytic. We achieve this by introducing *regulator*. *Regulator* is built atop existing vendor IoT gateway SDKs and enables flexible device configuration and data management, dynamic cloud service subscriptions and message routing. To realize the *ECIoT* vision, we designed the *regulator*, an ECIoT gateway subsystem that manages communication between vendor-specific IoT gateways and devices, and their respective cloud services. Evaluation results show that *regulator* achieves these multi-vendor objectives with negligible overhead although this additional overhead may negatively impact low-latency applications.

2 PROPOSED ECIOT: DESIGN

Fig. 1 summarizes the high-level architecture of the ECIoT gateway. At a high level, our design introduces a wrapper dubbed *Regulator*, which controls the communication links to the cloud-based applications. We further leverage current vendor gateway SDKs to connect to their IoT cloud portals enabling edge-functionalities.

Runtime: In our design, the runtime does the heavy work. As our goal is to build atop existing IoT gateways, our design leverages existing vendor gateway runtimes. AWS GG and Azure IoT gateway

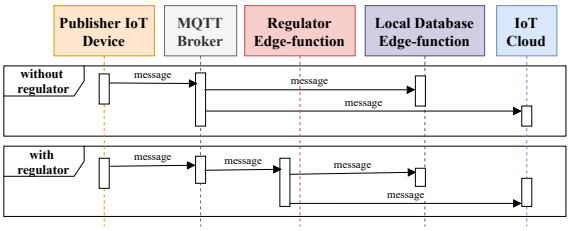

Figure 2: Sequence diagram of message flow in ECIoT

runtimes [2, 8] come pre-build with a task scheduler, and an MQTT Broker (for receiving IoT data from IoT edge devices and forwarding to the their clouds). We therefore use both runtimes during our implementation and show evaluation results in sec. 5. We further augment their runtimes with *Regulator*. We present a detail overview of *Regulator* later in this section.

IoT gateway SDKs: Within our edge-centric IoT gateway, we run Google, Azure and AWS gateway SDKs [4, 7, 8] as edge-functions. They provide the RESTFul APIs to communicate with their IoT cloud portal. However, as describe later, *regulator* controls all paths in our design bringing IoT data closer to the edge. We host a local database for data storage and a web server for local configurations and management, which in turn alleviates the cloud-centric management. Both the web sever and database run as edge-functions.

***Regulator* - Edge-function:** In our gateway design, *regulator* is primarily controlled by user configuration via the local webserver. Specific configuration options like "disable publish to cloud", "delete path x" and "create path y" can be configured. We consider "disable publish to cloud" during our implementation to show preliminary results later-on.

Traditionally, performing this function requires manually deleting and redeploying the configuration locally on the gateway on premise. Unlike current gateways, *ECIoT's* webserver edge-function performs a runtime interrupt via *regulator*. (i) *Regulator* leverages the runtimes' exposed APIs to discover the current static paths (source, destination, topics) pairs. (ii) It creates (if it does not exist) a new "shared topic" and subscriber (usually the local database) on the system and (iii) temporally disables the cloud facing path and redirects every packet via the new "topic"(path).

3 IMPLEMENTATION AND EVALUATION

Our experiment setup consists of an IoT Gateway, simulated IoT devices and their respective vendor-specific cloud portals. We used a Raspberry Pi2 to run *ECIoT*. We use both AWS and Azure IoT Gateway runtimes in our evaluation. We ran simulated IoT devices to send MQTT messages to *ECIoT* periodically. We didn't use any server instances in the cloud for the setup other than the basic MQTT broker and the MQTT cloud test utility provided by the respective IoT hub frameworks to view the MQTT messages on the cloud.

We implemented four edge-functions: a web-server, *regulator*, a local database and Google IoT SDK. The web-server serves web-api's along with web pages for the various local configurations of *ECIoT*. The local database is used to store IoT data. To enable multi-vendor CSPs communication, we configured the Google IoT device SDK as an edge-function in both AWS and Azure runtimes. Since the necessary security mechanisms are already implemented within the SDK, the Google SDK edge-function is able to authenticate and communicate with the Google cloud (Google IoT Core) [6]. These communication is controlled and managed by *regulator* as discussed next.

During the start-up sequence of the *regulator*, a local message subscription configuration file is loaded with the list of message topics or endpoints and a flag to indicate whether the message should be forwarded to the cloud or not. Changes to this local configuration file is done via the local web server. Even though there may be a message subscription configured in the cloud, *regulator* overrules these configurations and controls the traffic based on the local user configuration. Fig. 2 shows how the MQTT message paths are altered with *regulator*. Through this implementation, we are able to enhance the existing IoT Gateway frameworks to support local control of the message subscriptions regardless of cloud configurations and enable local control of the message routes between vendor-specific IoT Gateways and multi-vendor IoT Hubs, which is not supported in traditionally IoT Gateways.

3.1 Evaluation

In this section, we seek to understand the performance degradation incurred in our proposed design. In Fig. 2, we show the sequence diagram of the packets flow in our proposed gateway with and without *regulator*. We can see that, once *regulator* is introduced into the system, every packets from the mqtt broker is routed via *regulator*. The traffic is then steered based on the local user configuration. *Regulator* serves as a wrapper to enable dynamic subscriptions and interoperate between different vendor platforms.

On the gateway, we collect the CPU and memory utilization with and without *regulator*. Fig 3 and 4, both show the CPU (top plot) and RAM usage (bottom plot) performance metrics collected. Observe that, as expected, augmenting both AWS GG and Azure runtimes

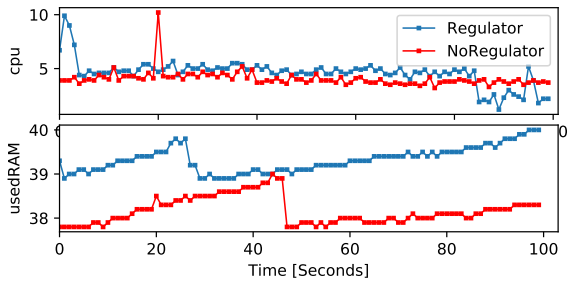

Figure 3: Azure runtime memory and cpu utilization

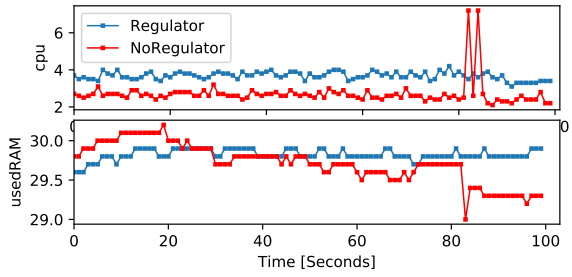

Figure 4: AWS runtime memory and cpu utilization

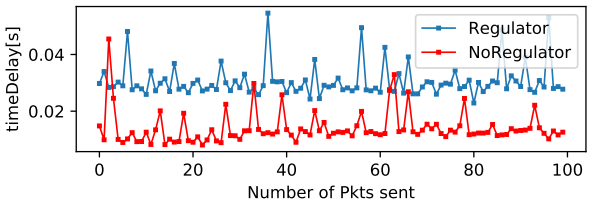

Figure 5: AWS runtime time delay

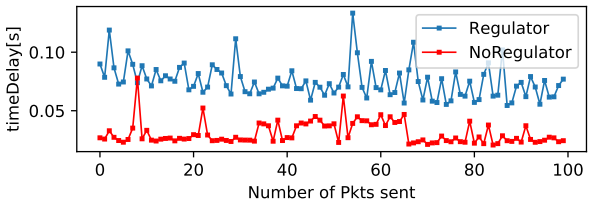

Figure 6: Azure runtime time delay

with *regulator* (the blue line) result in more CPU and Memory usage. This is because, all packets through the gateway is processed by *regulator*, *i.e.,* the packet header information needs to matched the paths deployed on the system before routing.

Next, we want to quantify the additional delay incurred by processing every packets with *regulator*. We simulated IoT devices to publish data to *ECIoT* and logged the time when every packet is sent. We configure a path to route every packet to the local database, where we log the time every packet is received. We repeat this experience with and without *regulator* deployed. In Fig. 5 and 6, we show the delay incurred with *regulator* (the blue curve) and without *regulator* (the red curve) leveraging both AWS GG runtime and Azure runtime respectively. As observe, *regulator*, as a wrapper incurs negligible overhead while still supporting current vendor SDK gateways.

4 CONCLUSIONS AND FUTURE WORK

To conclude, we proposed a shift from a Cloud-centric to an Edge-centric approach to IoT. We introduced *regulator*, which augments current vendor locked IoT platform solutions and controls the paths from gateway SDKs to the cloud. Evaluations of our experiments show that our approach incurs an additional negligible overhead and minimal latency. Nonetheless, we acknowledge that this additional overhead may negatively impact low-latency applications in certain IoT scenarios. This study extenuates that possible research directions can be 1) to address the interoperability issues with vendor edge devices and 2) build an IoT gateway runtime to support heavy edge computational tasks and connect to multiple vendor cloud platforms.

5 ACKNOWLEDGEMENT

The research was supported in part by NSF under Grants CNS-1618339, CNS-1617729, CNS-1814322, CNS-1836722 and CNS-1901103.

REFERENCES

[1] Google Cloud . 2019. Overview of Internet of Things | Solutions | Google Cloud. (02 2019). https://cloud.google.com/solutions/iot-overview
[2] AWS. [n. d.]. Configure devices and subscriptions. ([n. d.]). https://docs.aws.amazon.com/greengrass/latest/developerguide/config-dev-subs.html
[3] AWS. 2018. Run Lambda functions on the AWS IoT Greengrass core - AWS IoT Greengrass. (11 2018). https://docs.aws.amazon.com/greengrass/latest/developerguide/lambda-functions.html
[4] AWS. 2019. Machine Learning Inference with AWS IoT Greengrass Solution Accelerator. (10 2019). https://aws.amazon.com/iot/solutions/mli-accelerator/
[5] Sharu Bansal and Dilip Kumar. 2020. IoT Ecosystem: A Survey on Devices, Gateways, Operating Systems, Middleware and Communication. *International Journal of Wireless Information Networks* (2020), 1–25.
[6] Google Cloud. 2019. Publishing over the MQTT bridge | Cloud IoT Core Documentation. (2019). https://cloud.google.com/iot/docs/how-tos/mqtt-bridge
[7] Google. [n. d.]. Using gateways. ([n. d.]). https://cloud.google.com/iot/docs/how-tos/gateway
[8] Microsoft. [n. d.]. Configure an IoT Edge device to act as a transparent gateway. ([n. d.]). https://docs.microsoft.com/en-us/azure/iot-edge/how-to-create-transparent-gateway