Towards a Proactive Lightweight Serverless Edge Cloud for Internet-of-Things Applications

Ian-Chin Wang, Shixiong Qi, Elizabeth Liri, K. K. Ramakrishnan
University of California, Riverside

2021 IEEE International Conference on Networking, Architecture and Storage (NAS) | 978-1-7281-7744-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/NAS51552.2021.9605384

Abstract—Edge cloud solutions that bring the cloud closer to the sensors can be very useful to meet the low latency requirements of many Internet-of-Things (IoT) applications. However, IoT traffic can also be intermittent, so running applications constantly can be wasteful. Therefore, having a serverless edge cloud that is responsive and provides low-latency features is a very attractive option for a resource and cost-efficient IoT application environment.

In this paper, we discuss the key components needed to support IoT traffic in the serverless edge cloud and identify the critical challenges that make it difficult to directly use existing serverless solutions such as Knative, for IoT applications. These include overhead from heavyweight components for managing the overall system and software adaptors for communication protocol translation used in off-the-shelf serverless platforms that are designed for large-scale centralized clouds. The latency imposed by 'cold start' is a further deterrent.

To address these challenges we redesign several components of the Knative serverless framework. We use a streamlined protocol adaptor to leverage the MQTT IoT protocol in our serverless framework for IoT event processing. We also create a novel, event-driven proxy based on the extended Berkeley Packet Filter (eBPF), to replace the regular heavyweight Knative queue proxy. Our preliminary experimental results show that the event-driven proxy is a suitable replacement for the queue proxy in an IoT serverless environment and results in lower CPU usage and a higher request throughput.

Index Terms—Internet-of-Things, serverless, edge cloud

I. Introduction

Internet-of-Things (IoT) solutions have been increasingly adopted for having universally connected sensors and actuators that can be accessed from anywhere. IoT applications that process the sensor data, however, are subject to a number of requirements and constraints, e.g., low latency, limited capital expenditure and operational cost, low power, etc. An edge cloud for IoT services reduces the network latency since it can physically be closer to the sensor devices [1]. However, the intermittent nature of IoT traffic also suggests the need for an efficient low-cost backend to host the service on-demand. Only using server resources when the application function is invoked has the potential to achieve very good statistical multiplexing and thus lower cost. Thus serverless computing is very well suited for IoT environments, due to the promise to handle intermittent traffic. The cloud's scalability helps support bursts of traffic, as IoT traffic intensity can also be highly variable.

A general problem in serverless computing is the "cold start". This occurs when traffic arrives at the service backend and no computing instance is ready, so the packets have to wait till the service is started. This results in excessive response latency. Most cloud orchestration frameworks, including serverless computing, can take seconds (even minutes) to bring up a service, depending on its complexity. But, depending on the application scenario, the acceptable delays for IoT applications and users can also widely vary [2]. For example, in an IoT parking management application, it may not be a major impediment if the number of available spaces is not updated immediately (within a second) after a vehicle exits out of the parking space, given the frequency of car arrivals, especially if the structure is not full. However, a delay of even a few hundred milliseconds may result in a serious accident in an autonomous driving application.

We focus on Knative [3], a general-purpose open-source serverless platform with a function chaining solution that supports IoT applications. The standard Kubernetes cloud orchestration and Knative serverless framework are built to be more suitable for large centralized cloud environments that are 'resource rich'. Their design approach emphasizes flexibility and reuse of existing components and frameworks. As a result, they can be quite inefficient for a resource-constrained edge cloud and for use in cost-sensitive IoT applications.

Similar to most service platforms, Knative adopts HTTP to support communication among its services. IoT devices, due to their constrained energy resources, adopt lightweight communication protocols, e.g., Constrained Application Protocol (CoAP) [4], Message Queuing Telemetry Transport (MQTT) [5], to limit the overhead of protocol headers and the amount of messaging to match the often small amount of payload to be communicated. In this paper, we primarily focus on the MQTT protocol and the support needed for it in the serverless edge cloud. However, our design is broadly applicable to other IoT applications and protocols.

A protocol conversion from MQTT to HTTP is needed before microservices running on Knative can process IoT traffic. This conversion is typically done through a general-purpose MQTT-to-HTTP adaptor, e.g., an Apache Camel [6] plugin, which is called MQTT Source [7]. This plugin is not optimized to serve in the resource-constrained edge IoT scenario. Directly deploying this standard MQTT Source protocol adaptor to handle IoT events results in an individual topic-specific protocol adaptor instance, since the functions are organized on the basis of different MQTT topics. Such an implementation incurs substantial, overhead. There is a clear need for one single instance of the protocol adaptor to demultiplex requests across all the functions.

978-1-7281-7744-1/21/$31.00 ©2021 IEEE
Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on June 29,2023 at 18:39:28 UTC from IEEE Xplore. Restrictions apply.

There is additional overhead in Knative due to the sidecar container (i.e., queue proxy). In Knative, each function instance (i.e., pod) has a dedicated sidecar container to facilitate HTTP networking of the function instance. This sidecar container is a continuously running process throughout the pod's lifetime, consuming CPU resources. While it is acceptable for web services with a high number of requests, it is less suitable for IoT traffic which may be infrequent and intermittent. Further, the hardware resources available at the edge cloud are likely to be more limited compared with centralized data centers. Thus, the edge cloud may find it challenging to support a large number of sensors. Moreover, service providers may be required to cover the cost of the wasted computing resources when supporting intermittent IoT traffic.

In this paper, we address the challenge of supporting IoT applications with the typical Knative platform and present a number of optimizations for a serverless IoT framework for an edge cloud environment. We consider the case where each IoT sensor transmits its data using the MQTT protocol to the Knative serverless edge cloud. This sensor data is usually identified by a topic and is processed by services subscribed to that topic. We enhance the Knative-Apache Camel framework to receive MQTT traffic across all the topics with one lightweight MQTT-to-HTTP adaptor instance to handle all scenarios, including multiple types of sensors.

Another heavyweight component in Knative is a sidecar called the queue-proxy, which aids in pod metric collection for autoscaling. Instead, we propose the use of an event-driven eProxy that runs as an eBPF program to replace the queue proxy. To minimize cold start latency, our design proposes using traffic prediction in conjunction with a modified autoscaler that provisions and pre-warms the function pods appropriately before traffic arrives or ramps up.

We present evaluation results for the CPU usage and request processing delay, for different system configurations, to supporting MQTT traffic from a synthetic traffic generator. For comparison purposes we also show the performance with HTTP-based IoT traffic handled by Knative, to understand the performance penalty for MQTT. We also present the results with our event-driven 'eproxy' that significantly reduces the high resource utilization in the Knative design.

II. Related Work

Mohanty et al. [8] and Li et al. [9] evaluate the performance of open-source serverless platforms and considered the response time and ratio of successfully received responses, etc. to understand various design choices. Lloyd et al. [10] identify four states of the serverless hosting infrastructure. They then demonstrated that depending on the state, the microservices performance can vary up to 15.

The advantages of microservices, functions on-demand, and auto-scaling make it a promising candidate to handle the processing needs of various types of IoT traffic. However, resource management remains a key concern and one way to address this is to understand traffic patterns and then proactively provision resources. For example, Mazhar et al. [11] performed a measurement study in home environments and real-world logs collected from more than 200 homes in a metropolitan area to characterize smart home IoT traffic.

III. Overview of the Design

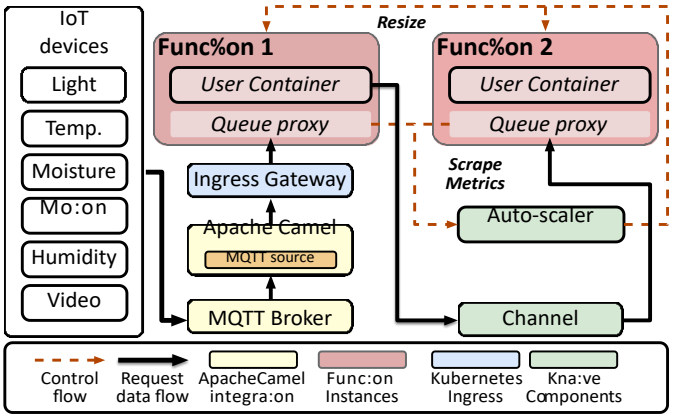


Fig. 1. Knative-based serverless edge cloud for IoT applications

A. Overview

Knative is an open-source Kubernetes-based serverless platform allowing dynamic management of serverless functions [3]. It facilitates cloud service provisioning and management by proactively adjusting the services deployed. In a regular Knative-based IoT serverless cloud setup, various components coexist to support sensor data processing, as shown in Fig. 1. We used the Apache Camel as the protocol adaptor to convert the MQTT-based IoT data into HTTP-based cloud events, which can then be consumed by the serverless function chains built around the serverless middleware for handling requests and responses.

Some challenges still exist in the Knative-based framework. By default, the autoscaler only scales the functions after the traffic arrives, which increases the service response time because of queuing at under-provisioned serverless instances or cold start latency if the system is currently operating at 'zero-scale' (no active function instances). We propose an XGBoost based traffic prediction solution to minimize the cold start latencies. In addition, there are several heavyweight general-purpose components. The CPU processing load of the Knative queue proxy is significant and it is continuously running, independent of the traffic. Apache Camel, is designed as a generic event delivery middleware. It requires a dedicated instance to serve each MQTT topic, even if some MQTT topics are idle most of the time, thus causing unnecessary CPU overhead as well.

In order to implement a Knative-based IoT serverless edge cloud, we designed a lightweight MQTT adaptor to replace the existing design which has a high overhead for inputs from multiple sensor types. Importantly, we design an event-driven proxy based on eBPF replacing the continuously-running queue proxy for each function pod, thereby reduce the significant processing overhead.

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on June 29,2023 at 18:39:28 UTC from IEEE Xplore. Restrictions apply.

B. Predictor and Proactive Scaling

[12] shows that a simple single layer perceptron prediction mechanism and function pre-warming helps mitigate the cold start delay. This would allow function pods to already be available when the actual IoT traffic arrives. We designed a low-complexity predictor to anticipate future arrivals using an XGBoost [13] to improve this. However, this prediction component of our design is not evaluated in this paper. We only report results with synthetic workloads generating a steady traffic rate. Future work includes the evaluation of this component using multiple real-world workloads.

C. MQTT Broker and HTTP Adaptor

In Knative, the default event handling mechanism for MQTT traffic treats each MQTT channel as an independent event source. The Apache Camel middleware then aggregates the messages from the MQTT broker with a dedicated MQTT Source for each MQTT channel to transform the MQTT requests into HTTP. We use Mosquitto [14] as the MQTT broker to direct the data flow from the IoT sensors to the serverless function chains. The Mosquitto broker is relatively lightweight (as we observe in our experiments) and accepts traffic from multiple publishers (sensors). The MQTT broker runs continuously to provide low latency. It can also be horizontally scaled by Kubernetes, based on the incoming traffic rate. Although the adaptor obtains the service DNS name/IP address, sending the message to only one subscribing instance/function pod is handled by the Kube-proxy which selects a specific destination pod IP in a round-robin manner.

D. Event-driven proxy (eProxy)

Since the Knative queue proxy is a major source of overhead [12], it is not ideal for a serverless IoT environment. We replace the Knative queue proxy (which provides a readiness check and does the metrics collection), by other components that collectively implement its functionality. These components either are already working as indispensable processes (e.g., kubelet) or have been integrated into our event-driven eBPF program, which we call the eProxy. The kubelet, which is one instance running on the entire host, uses HTTP probing to check the readiness of the function instances periodically. The eProxy is only triggered when there are incoming events (e.g., packets). Thus, there is no CPU overhead in the idle state and it is perfectly suited to meet the high-efficiency requirement of serverless IoT. The eProxy is attached to the pod's virtual Ethernet interface (veth) to collect metrics (e.g., the number of requests, the response latency). In collaboration with the in-kernel persistent storage (i.e., an eBPF map) and the user-space metrics exporter, the eProxy is able to report the per-pod metrics to the control plane. In the control plane, the autoscaler leverages the metrics and scales the pod instances up/down as needed for the IoT service. However, dynamically loading the eProxy code increases the startup time for a serverless function [15], interfering with our efforts to mitigate the cold-start latency. This requires further improvement, which is a focus of our current work.

TABLE I
CPU OVERHEAD BREAKDOWN

Components	User cont.	Queue Proxy	Adaptor	Broker	Other
HTTP-QPROXY	6.53	7.07	NULL	NULL	1.50
HTTP-EPROXY	6.07	NULL	NULL	NULL	1.92
MQTT-QPROXY	6.59	6.19	1.92	0.83	1.24
MQTT-EPROXY	5.88	NULL	1.77	0.88	1.45

IV. EVALUATION & ANALYSIS

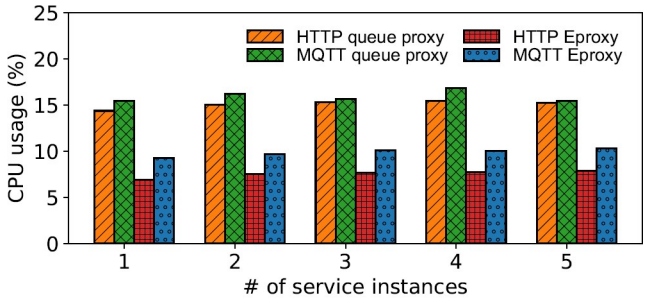
Due to space limitations, we primarily focus on evaluating the eProxy, MQTT broker, and adaptor with a synthetic workload. The experiment setup had two physical nodes connected by a 25 Gbps link. The workload generator was deployed on the master node and all the IoT environment building blocks, including the service instances, MQTT broker, MQTT-to-HTTP adapter, etc. were placed on a worker node. We ran the experiments with two different service protocol modes: HTTP mode and MQTT mode. Both modes work in conjunction with two different sidecar proxy support components (i.e., Knative queue proxy and eProxy), which results in four different modes namely HTTP-QPROXY, MQTT-QPROXY, HTTP-EPROXY, MQTT-EPROXY. In HTTP mode, the request generator sends HTTP requests directly to the service instances. In MQTT mode, the MQTT workload generator sends the requests to the MQTT broker which then forwards them to the adapter. The MQTT requests are converted to HTTP messages by the MQTT-to-HTTP adaptor and then sent to the target service. We used Apache benchmark [16] as the HTTP workload generator and built a custom MQTT workload generator based on a Paho Python Client [17].

Since all the IoT environment building blocks were placed together on the worker node, we measured the overall CPU usage of the worker node to compare the resource overhead between different modes. We also measured the delay per request to understand the impact of MQTT and eProxy on the datapath. The measurement experiments were performed while varying the number of service instances. In each experiment, we kept the request per second (RPS) at approximately 1K[1], which ensured the IoT building blocks under different modes can receive similar traffic loads. We show the CPU overhead breakdown of individual components for different modes.

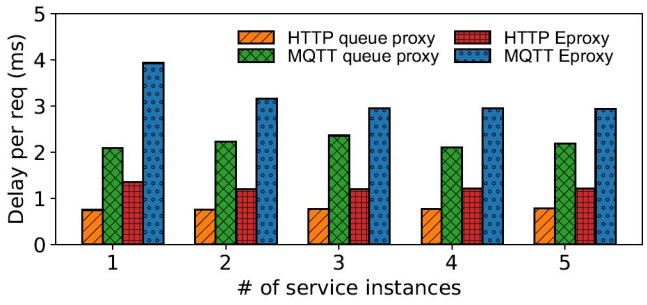
For the workload intensity we tested, for the MQTT-QPROXY mode, as shown in Table I, the queue proxy consumes 6.19% of the overall CPU usage, which is very close to the user-container (6.59%). This high cost for queue proxy makes it undesirable in a resource-constrained serverless edge cloud. Examining the MQTT-EPROXY mode, there is a significant reduction in total CPU usage when we replace the queue proxy with the eProxy. As shown in Fig. 2(a), the eProxy can save up to 37% CPU consumption compared

[1]Note: We sought to maintain the RPS as close to the target value as possible. The generator maintained the maximum number of outstanding requests, sending a new request after receiving an acknowledgment from the service. Thus, the RPS of 1K was a slight approximation, with the value varying slightly over time.

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on June 29,2023 at 18:39:28 UTC from IEEE Xplore. Restrictions apply.



(a) CPU usage



(b) Delay per request

Fig. 2. CPU usage and request processing delay with varying # of instances

to the queue proxy, which is a result of the event-driven feature of the eProxy. The CPU consumption of the eProxy is relatively small and is captured in the slight increase of the 'other' column (along with the other components in the worker node) in the MQTT-EPROXY mode row (same with the HTTP mode). Our use of the Mosquitto broker and the Camel-based MQTT Source adaptor appear to introduce a reasonable, manageable overhead. Using our enhancements (improved eProxy and adaptor), there is only a small increase difference in CPU processing for MQTT-EPROXY compared to HTTP-EPROXY (Fig. 2(a)). This should therefore not discourage the use of MQTT considering the client and networking benefits of MQTT for IoT environments.

Fig. 2(b) shows the delay per request for different experiment modes. eProxy adds more base latency compared to queue proxy for both the HTTP (0:47ms more) and MQTT (0:99ms more) mode. However, the CPU usage reduction with the eProxy can help considerably at higher loads when the queue proxy will become a bottleneck and start contributing an increasing amount of queue delay. The lightweight feature of eProxy makes it more suitable in the serverless edge cloud, especially for IoT. In addition, with the same sidecar proxy configuration, MQTT mode adds more latency compared to HTTP mode, because of the additional intermediate MQTT components, i.e., MQTT broker and MQTT-to-HTTP adaptor. We plan to understand why and seek to overcome this additional delay in the near future.

V. Conclusions

Serverless computing is well suited for IoT environments, where traffic can be infrequent and varying in intensity. We discussed the key components needed to support IoT traffic in an edge cloud and present a serverless framework to support such IoT traffic. We used Knative as the base for our serverless solution which primarily uses HTTP for client interaction. Since IoT devices generally use lightweight protocols such as CoAP and MQTT to reduce overhead and energy consumption, we need to incorporate an MQTT broker and a lightweight adaptor to translate MQTT requests to the HTTP requests that are usually processed by Knative functions. To reduce overhead, we replaced the queue proxy of Knative with an event-driven eBPF based "eProxy". Experimental results showed that our eProxy is a suitable replacement for the queue proxy and results in lower CPU usage.

Acknowledgments

We thank US National Science Foundation for their generous support through grants CNS-1619441 and CNS-1763929.

References

[1] J. Pan and J. McElhannon, "Future edge cloud and edge computing for internet of things applications," IEEE Internet of Things Journal, vol. 5, no. 1, pp. 439–449, 2017.
[2] J. Mocnej, A. Pekar, W. K. Seah, and I. Zolotova, Network traffic characteristics of the IoT application use cases. School of Engineering and Computer Science, Victoria University of Wellington, 2018.
[3] "Knative." [Online]. Available: https://knative.dev
[4] C. Bormann, A. P. Castellani, and Z. Shelby, "Coap: An application protocol for billions of tiny internet nodes," IEEE Internet Computing, vol. 16, no. 2, pp. 62–67, 2012.
[5] M. B. Yassein, M. Q. Shatnawi, S. Aljwarneh, and R. Al-Hatmi, "Internet of things: Survey and open issues of mqtt protocol," in 2017 international conference on engineering & MIS (ICEMIS). IEEE, 2017, pp. 1–6.
[6] "Apache Camel," https://camel.apache.org, 2021, [online].
[7] "MQTT Source," https://camel.apache.org/camel-kamelets/latest/mqtt-source.html, 2021, [online].
[8] S. Mohanty, G. Premsankar, and M. diFrancesco, "An evaluation of open source serverless computing frameworks," in 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2018, p. 115–120.
[9] J. Li, S. G. Kulkarni, K. Ramakrishnan, and D. Li, "Understanding open source serverless platforms: Design considerations and performance," in Proceedings of the 5th International Workshop on Serverless Computing, 2019, pp. 37–42.
[10] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in 2018 IEEE International Conference on Cloud Engineering (IC2E), 2018, pp. 159–169.
[11] M. H. Mazhar and Z. Shafiq, "Characterizing smart home iot traffic in the wild," in 2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI), 2020, pp. 203–215.
[12] I. Wang, E. Liri, and K. K. Ramakrishnan, "Supporting IoT Applications with Serverless Edge Clouds," in 2020 IEEE 9th International Conference on Cloud Networking (CloudNet). IEEE, 2020, pp. 1–4.
[13] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16. ACM, August 2016, pp. 785–794.
[14] R. A. Light, "Mosquitto: server and client implementation of the mqtt protocol," in he Journal of Open Source Software 2(13), May 2017.
[15] V. Jain, S. Qi, and K. Ramakrishnan, "Fast function instantiation with alternate virtualization approaches," in 2021 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN). IEEE, 2021, pp. 1–6.
[16] "ab - Apache HTTP server benchmarking tool," https://httpd.apache.org/docs/2.4/programs/ab.html, 2021, [online].
[17] "Eclipse Paho," https://www.eclipse.org/paho/, 2021, [online].

Authorized licensed use limited to: Univ of Calif Riverside. Downloaded on June 29,2023 at 18:39:28 UTC from IEEE Xplore. Restrictions apply.