2018 Third ACM/IEEE Symposium on Edge Computing

Extend Cloud to Edge with KubeEdge

Ying Xiong, ying.xiong1@huawei.com; Yulin Sun, yulin.sun@huawei.com; Li Xing, Li.xing1@huawei.com; Ying Huang, ying.huang@huawei.com;

Seattle Cloud Lab, Huawei R&D USA, Bellevue WA

ABSTRACT: In this paper, we introduce an infrastructure in edge computing environment, KubeEdge, to extend cloud capabilities to the edge. In the new form of cloud architecture, Cloud consists of computing resources both at centralized data centers and at distributed edges. KubeEdge infrastructure connects and coordinates two computing environments for applications leveraging both computing resources to achieve better performance and user experience. Technically, KubeEdge provides the network protocol infrastructure and the same runtime environment on the edge as in the cloud, which allows the seamless communication of applications with components running on edge nodes as well as cloud servers. It also allows the existing cloud services and cloud development model to be adopted at edge. Based on Kubernetes [1], KubeEdge architecture includes a network protocol stack called KubeBus, a distributed metadata store and synchronization service, and a lightweight agent (EdgeCore) for the edge. KubeBus is designed to have its own implementation of OSI network protocol layers, which connects servers at edge and VMs in the cloud as one virtual network. KubeBus provides a unified multitenant communication infrastructure with fault tolerance and high availability. The distributed metadata store and sync service is designed to support the offline scenario when edge nodes are not connected to the cloud. EdgeController component in KubeEdge architecture is a controller plugin for Kubernetes [1] to manage remote edge nodes and cloud VMs as one logical cluster, which enables KubeEdge to schedule, deploy and manage container applications across edge and cloud with the same API.

KEYWORDS; Edge Computing, Distributed Systems, Cloud Computing, Network Protocol and Data Synchronization.

I. Introduction and Related Work

With the rapidly growing requirements for edge based applications such as IoT, AI and stream data analytics, Edge Computing, which enables computation to be "performed at the edge of the network, on downstream data on behalf of cloud services and upstream data on behalf of IoT services"[2], becomes more and more important for cloud computing.

There are many researches and/or edge computing solutions used in various use case scenarios. For example, the platform [3] proposed by Hung Cao and Monica Wachowicz was deployed on mobile physical devices on the transit bus and is used to perform descriptive analytics on real-time transit data streams to uncover meaningful patterns. A recent research [4] examined and investigated computation partitioning strategies that effectively leverage both the cycles in the cloud and on the mobile device to achieve low latency, low energy consumption, and high datacenter throughput for Deep Neural Networks (DNNs) based machine learning intelligent applications. Their study shows that, by scheduling DNN computation between mobile devices and data centers, it can improve end-to-end latency by 3.1 times, reduce mobile energy consumption by 59.5%, and improve datacenter throughput by 1.5 times on average.

In the era of Internet of Thing (IoT), billions of sensors and actuators are deployed worldwide. To manage the IoT devices and process data with cloud computing resource, Cloud providers such as Amazon AWS and Microsoft Azure are developing the IoT platform and are providing services or solutions on their respective cloud environments. Most IoT platforms employ a Pub/Sub brokers such as MQTT [9] or AMQP to provide the communication channel between IoT devices and Cloud services, like Azure IoT Hub.

For end-to-end IoT solution, cloud providers also provide a component running on the devices or on the edge nodes that are close to the devices, such as AWS GreenGrass [5] and Azure IoT Edge [6]. These edge components manage the execution of local IoT applications and communication channel for data transfer to and from the cloud. For example, AWS GreenGrass extends the Lambda function environment to the edge and allows Lambda functions to be deployed and run on GreenGrass nodes, and Pub/Sub protocol is used for the communication between cloud and GreenGrass. The same is true for Azure IoT solution where Azure edge hub extends cloud runtime environment to IoT edge nodes which communicate with cloud services through Pub/Sub message protocol. Pub/Sub protocol such as MQTT is suitable for asynchronous communication between edge devices and cloud services. However it does not support synchronous RPC based communication, for which we have seen the increased need as more and more computation tasks [3][4] move to the edges and tightly integrate with services in the cloud.

One common scenario for RPC based communication is the cloud native micro-service based application. With micro-service architecture, an application is designed into multiple micro services, each of which is deployed and managed independently. These micro services communicate each other usually through REST/HTTP protocol. When some of the micro services run on the edge nodes and need to communicate with those in the cloud, it requires the one

DOI 10.1109/SEC.2018.00048

network address space for both edge nodes and server instances in the cloud. This is where current edge computing solutions break down, partly due to the asynchronous Pub/Sub based MQTT protocol. In addition to the communication protocol, to achieve the goal that any micro service can freely be scheduled to run on the edge or in the cloud, an edge platform needs to provide a unified runtime environment across devices, edge nodes and cloud servers. This paper presents an Edge infrastructure, called KubeEdge. The Infrastructure leverages Kubernetes container platform to provide RPC based communication channel between edge and cloud, the runtime execution environment of containers and Serverless functions, as well as a mechanism to sync and store metadata to support self-management of an application running on the edge in an offline scenario.

KubeEdge platform is being integrated into our public cloud as edge cloud service for evaluation. In this paper, we show some of preliminary experimental results and future planed works.

II. ARCHITETCURE OF KUBEEDGE

As shown in **Figure** 1, KubeEdge is a multi-tenant infrastructure platform for edge computing.

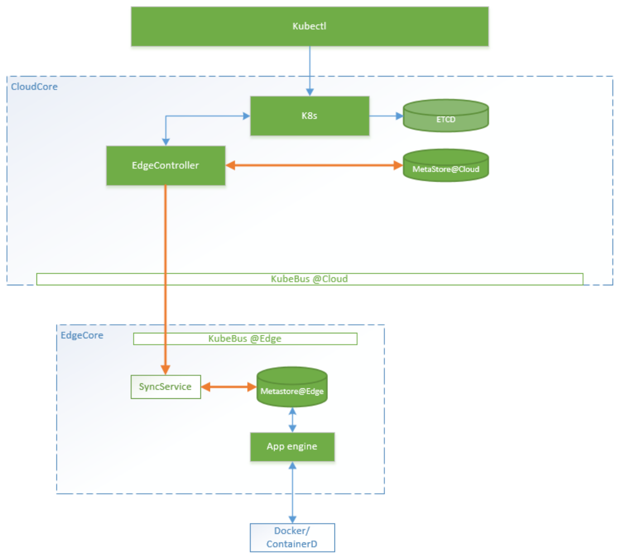

Figure 1: KubeEdge Architecture

The platform includes the following components, excluding Kubernetes.

1) KubeBus – A virtual network layer connecting edge nodes and cloud VMs as one addressable network space in a multi-tenant environment.

2) EdgeController – A Kubernetes controller plugin to enable KubeEdge (and Kubernetes) to remotely manage edge nodes as the cluster nodes, and allow applications or services to be deployed on the edge from the cloud through Kubernetes API.

3) MetadataSyncService – A bidirectional metadata sync services between edge and cloud for the platform itself and user applications.

4) EdgeCore – A lightweight agent running on the edge nodes to start up and manage container based applications as well as Serverless functions.

The following sections describe each component in detail.

A. KubeBus

KubeBus in the KubeEdge architecture is designed to address the network connectivity issue for applications running on the edge nodes connecting to cloud services, and vice visa. For example, a client video application running in the cloud can send http requests to a video streaming web service running on the edge nodes through KubeBus even though the edge nodes are physically in the private network (assume edge servers have Internet access). KubeBus component runs both in the cloud (KubeBus@Cloud) and at the edge (KubeBus@Edge), and it supports multi-tenancy, i.e., edge nodes and the applications running on these edge nodes can belong to different tenants where they share the same set of KubeBus instances running in the cloud.

1) Edge Node to Edge Node VPN

In a typical edge environment, an edge node is connected within a private local network without public IP address. Two edge nodes may be in two different private networks and they can't communicate each other. KubeBus solves the network issue by implementing L3 overlay network on top of cloud networking (for cloud VMs) and private local network (for edge nodes). **Figure** 2 below describes the implementation architecture of KubeBus. As shown in the diagram, KubeBus implements its own L2 and L3 over TCP connection. The data link layer in KubeBus@Edge establishes one or more long running TCP connection(s) to KubeBus@Cloud. The connection is directional duplex communication channel allowing either side to send requests or messages to another side between cloud and edge. In the case that two edge nodes in two different private networks need to communicate each other, one edge node sends the IP packet through KubeBus@edge to the KubeBus@Cloud and the packet is then routed by KubeBus@Cloud to another edge node via the already created long running TCP connection.

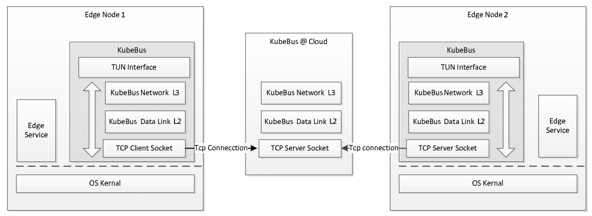

Figure 2: KubeBus Implementation Architecture

2) Connecting Edge Node to Cloud Network

The second network scenario supported by KubeBus is to connect an application running on the edge to the services running in the cloud virtual network.

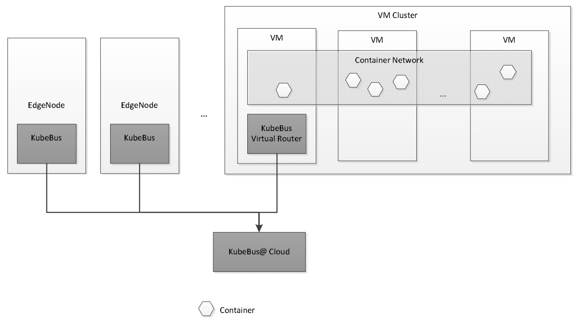

Figure 3: Connecting Edge VPN to Cloud Network

In this scenario shown in the **Figure** 3, two edge nodes are in their respective private subnet, created by KubeBus@Edge, and all VMs are in the VPC subnet created by cloud network. KubeBus then connects edges and VMs together as single VPN. This implementation includes a virtual router agent installed on one or more of the VMs in the cloud. The KubeBus virtual router agent contains the KubeBus network protocol stack and acts as a proxy between the edge subnet and the cloud VM subnet.

3) KubeEdge Http Protocol Stack

The following diagram (**Figure** 4) shows the KubeBus protocol stack for Http communication between edge and cloud.

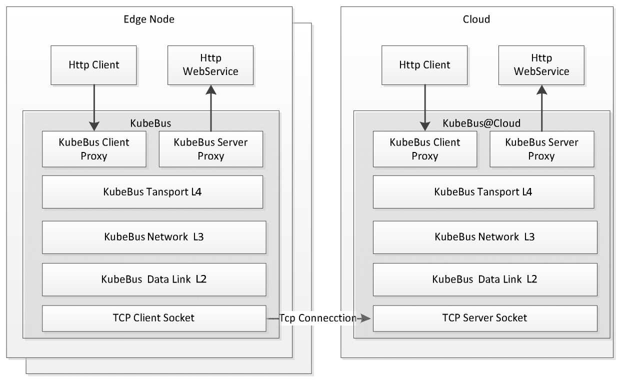

Figure 4: KubeBus Http Protocol Stack

In this protocol stack, transport layer L4 is a reliable connection layer built in KubeBus with fault tolerance. This layer provides the same interface as TCP, such as Listen, Accept, Connect and Disconnect APIs. On top of layer 4 in KubeEdge, there are two http reverse proxies implemented for direct http communication between edge and cloud. This is designed to support one common edge use case scenario where a video stream web service running on an edge node that is connected to a video camera locally, and users can watch the real-time video from web browsers through the http proxies at KubeEdge. The two http proxies are described as follows in detail:

a) KubeBus Client proxy listens on a TCP port for Http requests. Through service discovery, the proxy then forwards the requests to the corresponding KubeBus Server proxy at the same edge node or at another edge node, or in the cloud.

b) KubeBus Server proxy knows and manages the actual services running on an edge node or in a cloud VM. It forwards a request to the requested service, and returns the response back to the Client proxy where it gets the request.

To support multi-tenancy, each web service is registered to KubeBus with a globally unique identifier consisting of tenant id, edge node name and service name. KubeBus uses the global identifier as part of URL for forwarding and accessing the service, as shown below.

```
Http(s)://{hostname}/{Tenant Name}/{Edge Node Name}/{Http WebService Name}/…
```

B. EdgeController

In Kubernetes architecture [1], Kubelet runs as an agent on every node in a Kubernetes cluster. Kubelet watches Kubernetes master (API Server) through long running TCP connection for tasks to be performed at its node, such as starting, stopping and deleting application containers. Kubelet also reports the node and container status back to Kubernetes master. In the edge environment, edge nodes are far away from Kubernetes master running in the cloud, and the network connection may not be stable and bandwidth may be limited. EdgeController is designed to run in the cloud on behalf of edge nodes, i.e. it watches Kubernetes master for tasks for all edge nodes it represents. However, instead of actually performing these tasks on the edge nodes, EdgeController will send the metadata about the tasks to the corresponding edge nodes through KubeBus. The AppEngine module of EdgeCore component (to be discussed in section D) running on the edge node will actually perform the tasks assigned to the edge node. Thus, this design splits the normal Kubelet function into two parts, one running in the cloud, EdgeController and the other running on the edge, AppEngine. EdgeController is implemented as a controller plugin of Kubernetes so that it can watch Kubernetes master for tasks.

C. 3.2 MetadataSyncService

Metadata Sync Service, by its name, is responsible for synchronizing metadata between cloud and edge. The service is designed to solve the two issues:

1) To address the WAN reliable issue and to support self-management of the services running on the edge when the edge is disconnected from cloud (offline scenario)

2) To address the low bandwidth issue. In mant cases where the network bandwidth is limited at edge, we want to keep data size as minimum as possible for data transfer during synchronization. In other word, the incremental synchronization is preferred than the full snapshot synchronization.

Architecturally, the Metadata Sync Service includes two components, a metadata store and a synchronization service. Both run on the edge and in the cloud. The two instances work together to accomplish bidirectional data synchronization with fault tolerance. For example, when an edge node is offline, the write from cloud side will still succeed (writes will be stored in metadata storage). When the network connection is restored, the Sync Service will perform re-sync since last successful synchronization. The same is true for synchronization process from edge to the cloud. Sync Service supports atomic write and delta sync. The two data stores will be eventually consistent if the network connection between edge and cloud lost from time to time.

KubeEdge chooses Etcd [8] as the metadata storage, which supports transactional write and Multiple Version Concurrent Control (MVCC) API interface to retrieve the delta changes, i.e. Get/Watch based on Revision. The algorithm of data synchronization is shown in the following list (**Listing** 1)

```
Func SyncFromCloudToEdge(CloudStore, EdgeStore) {
        while true {
                LastSyncRevision = ReadLastSyncRevision(EdgeStore)
                Changes, NewRevision = ReadDelta(CloudStore, LastSyncRevision)
                TransactionalWrite(EdgeStore, Changes, NewRevision)
        }
}
```

Listing 1: Synchronization Algorithm

The algorithm is a simple loop, which performs three steps. First step is to retrieve the Last Sync Revision (LSR) of last successful synchronization from local etcd store, and the second step is to calculate delta changes and the new revision number. In last step, the algorithm performs a transactional write to local etcd store and the changes will be sync to the remote etcd store through etcd sync capability. The new revision will become the LSR in next loop iteration.

D. EdgeCore

EdgeCore is a lightweight agent running on every edge node that are registered with KubeEdge platform. It packages all KubeEdge functionality on the edge into one process. It includes KubeBus and MetadataSyncService, AppEngine and local etcd store.

III. PRELIMINARY EXPERIMENT

.

KubeEdge platform is being integrated with our public cloud as the edge cloud service. The platform has been evaluated in a test environment shown in **Figure** 5. In the diagram, Kubernetes master (KubeMaster) and two nodes (VM1/VM2) run in its container network in the cloud, and two edge nodes (Edge1 and Edge2) each run in its own subnets. All of these components are connected together through KubeBus via Internet. The KubueBus@Cloud run in one CentralVM in AWS cloud.

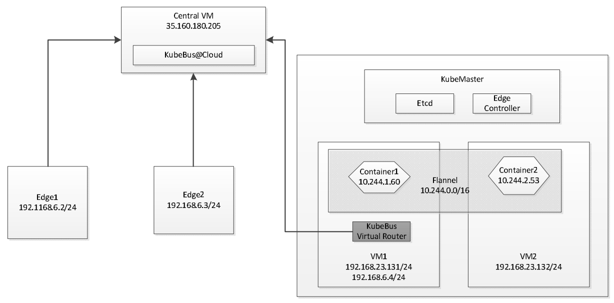

Figure 5: KubeEdge Test Environment

The test server configurations are specified in the following table (**Table** 1).

Server	Configuration
VM1	Amd64,2 core, 4G memory, Ubuntue 16.04
VM2	
Edge1	Respberry Pi 3 Model B, 1G memory, Quad Core 1.2GHz Broadcom BCM2837 64bit CPU, Linux raspberrypi 4.14.0-v7+
Edge2	Amd64,2 core, 4G memory, Ubuntue 16.04
CentralVM	Amd64,1 core, 1G memory, Ubuntue 16.04

Table 1: Server Configuration in Test

The initial evaluation is mainly focused on the network latency between edge and cloud for assessing the potential latency issues when edge nodes and applications are managed from the cloud. The table below (**Table** 2) shows the preliminary results.

Connection	Latency (ms)
Edge node 1 to Cloud	30.66
Edge node 1 to Edge node 2	61.24
Edge node 1 to VM 1 in the cloud	56.86
Edge node 1 to VM 2 in the cloud	55.88
Edge node 1 to Container 1 in VM1	60.68
Edge node 1 to Container 2 in VM2	66.78

Table 2: Communication Latency between Edge and Cloud

The results show that KubeEdge platform does not contribute to the normal network latency in a significant way. Further, we tested and compared the latency of deploying a container from KubeMaster to VM, and from KuberMaster to edge node (the container image download is not included in the test). The result shows that the container deployment latency from KubeMaster to VM#1 is about 2 seconds and the container deployment latency from KubeMaster to Edge node #1 is around 3 seconds, which is at acceptable level.

IV. CONCLUSION AND FUTUREWORK

In this paper, we presented an architecture based on Kubernetes platform for edge computing. We showed that it is possible for Kubernetes to manage remote edge nodes and deploy and manage applications into the edge with the same API. We implemented a communication protocol to allow seamless bidirectional communication between cloud and edge for applications leveraging both edge resources and cloud. We extended Kubernetes model from centralized data center deployment into the edge with EdgeController and EdgeCore components. This work is a preliminary experience and we intend to optimize and enhance KubeEdge as future work. One future work we consider is the concept and design of edge mesh network. In the current communication model, the inter-edge communication is through KubeBus at the cloud side. We have seen the scenarios that multiple edge nodes need to communicate each other directly without cloud, mainly due to local private network policies and data security. With the direct connections among edge nodes, KubeEdge will form a virtual network mesh among all edge nodes, and applications and services can leverage computing and storage resources on the other edge nodes. Another potential future work is the intelligent scheduling for edge applications based on data locality, network status and computing power.

REFERENCES.

[1] "Kubernetes: Production-Grade Container Orchestration", https://kubernetes.io/

[2] Quan Zhang, Xiaohong Zhang, Weisong Shi, "Firework: Big Data Processing in Collaborative Edge Environment", 2016 IEEE/ACM Symposium on Edge Computing (SEC).

[3] Hung Cao and Monica Wachowicz, Sangwhan Cha: Developing an edge computing platform for real-time descriptive analytics. 2017 IEEE International Conference on Big Data (Big Data) 11-14 Dec. 2017.

[4] Y Kang, J Hauswald, C Gao, A Rovinski, T Mudge, J Mars, L Tang, "Collaborative Intelligence Between the Cloud and Mobile Edge", CM SIGPLAN Notices 52 (4), 615-629.

[5] "AWS Greengrass Document", https://docs.aws.amazon.com/greengrass

[6] "Azure IoT Edge", https://azure.microsoft.com/en-us/services/iot-edge/

[7] "OpenVPN: Your private path to access network resource and service securely", https://openvpn.net/

[8] "etcd: A distributed, reliable key-value store for the most critical data of a distributed system", https://coreos.com/etcd/

[9] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "Mqtt-s -a publish/subscribe protocol for wireless sensor networks," in Communication systems software and middleware and workshops, 2008