Hindawi
Wireless Communications and Mobile Computing
Volume 2019, Article ID 3947286, 17 pages
https://doi.org/10.1155/2019/3947286

Research Article

Software-Defined Edge Cloud Framework for Resilient Multitenant Applications

Kaikai Liu,[1] **Nagthej Manangi Ravindrarao**,[1] **Abhishek Gurudutt**,[2] **Tejeshwar Kamaal**,[2] **Chinmayi Divakara**,[2] **and Praveen Prabhakaran**[2]

[1] *The Department of Computer Engineering, San Jose State University (SJSU), San Jose, CA, USA*
[2] *San Jose State University, USA*

Correspondence should be addressed to Kaikai Liu; kaikai.liu@sjsu.edu

Received 23 June 2018; Accepted 8 November 2018; Published 1 January 2019

Academic Editor: Yu Chen

Edge application's distributed nature presents significant challenges for developers in orchestrating and managing the multitenant applications. In this paper, we propose a practical edge cloud software framework for deploying multitenant distributed smart applications. Here we exploit commodity, a low cost embedded board to form distributed edge clusters. The cluster of geo-distributed and wireless edge nodes not only power multitenant IoT applications that are closer to the data source and the user, but also enable developers to remotely deploy and orchestrate application containers over the cloud. Specifically, we propose building a software platform to manage the distributed edge nodes along with support services to deploy and launch isolated and multitenant user applications through a lightweight container. In particular, we propose an architectural solution to improve the resilience of edge cloud services through peer collaborated service migration when the failures happen or when resources are overburdened. We focus on giving the developers a single point control of the infrastructure over the intermittent and lossy wide area networks (WANs) and enabling the remote deployment of multitenant applications.

1. Introduction

In recent decades, cloud computing has emerged in the information and communication world, providing various types of services, storage, computing, and networking. Data is aggregated to cloud data stores for intelligent processing with unlimited resources; applications are delivered and updated to data centers in real time; and computing infrastructure and resources can be shared with other applications on the go without interference. However, the main cloud is hosted in core data centers, which are sometimes far from the end user. This may introduce a considerable delay for end users' applications, thus preventing the deployment of services that are sensitive to latency, for example, cyber-physical-systems (CPS), smart cities applications, security monitoring, connected vehicles for Intelligent Transport Systems (ITS), or Internet of Things (IoT) [1–3]. Low latency analytics and real-time response are especially important for cyber-physical-system (CPS) applications. The dominant approach of aggregating all data to the data center stresses communication links, thus inflating the timeliness of analytics. Moving much of the processing to the locations where the event is happening facilitates real-time response and low communication overhead. To address the problem of the long latency, cloud services should be moved more proximally to the edge of mobile network [4, 5]. There is need for a system that can proactively provide computing services for these CPS and IoT devices as needed.

Mobile edge computing (MEC) can be understood as a specific case of mobile cloud computing (MCC), where the computing/storage resources are supposed to be in proximity to the user equipment (UE) [6, 7]. Hence, MEC can offer significantly lower latencies and jitter when compared to the MCC. Consequently, mobile edge computing may be considered one of the key enablers of Internet of Things (IoT) as it offers (1) low latency combined with location awareness due to proximity of the computing devices to the edge of the network, (2) widespread geographical distribution when compared to the data center; (3) interconnection of a very large number of nodes (e.g., wireless sensors); and (4) support

of streaming and real-time applications [8]. Many organizations contribute to the concept of edge computing under different initiatives, for example, the mobile edge computing (MEC) initiative by ETSI, Fog computing by Cisco [9, 10], and cloudlet by Carnegie Mellon University [11, 12]. Nevertheless, MEC also imposes huge additional challenges, including (1) multitenant support [13]; (2) automatic orchestration; and (3) resilience and availability.

(1) Multitenant Support. The support for multiple customers to use a single instance of the software infrastructure has been evolving from decades. We are leveraging the conceptual model of multitenant software architecture which is to run a single instance of software on a server and serving multiple tenants. The group of end customers/clients requiring the services of this edge cloud framework is called a tenant. These tenants share common access policies and have dedicated privileges for accessing our edge framework. Each tenant is provided with a shared instance including data storage, configuration and user management, and other preferences. With this type of software architecture for IoT models, we are utilizing data aggregation benefits through acquiring data from centralized storage instead of different database schemas, and the cost to implement multitenancy can also be minimized as a single instance is being used by many customers which substitutes the invested amount for memory and processing overhead.

IoT and CPS application developers need an infrastructure to support multitenant applications. Several of the existing edge computing platforms support either a single-user application, an application specific cloudlet, or a single service-oriented cloudlet. To operate, maintain, and secure this edge cloud network, researchers must grapple with multiple vendor-specific computing and sensing modules to implement complex high-level management policies. For example, multiple companies are applying to install smart sensors in street lights. However, this can be rather slow and labor-intensive due to the administrative process of determining which companies to offer permission. After the nodes are installed, they become the private property of the vendor. That said, it is difficult for new applications and services to utilize this existing platform. Similar to cloud computing, the computing and sensing modules available in edge nodes should be virtualized as a resource pool that integrates a cluster of applications to provide agility, responsiveness, and less overhead than traditional hypervisor-based virtualization. Despite many previous proposals to make the VMs in data centers easier to manage, many approaches are not fit for this edge cloud scenario or only amount to stop-gap solutions because of the underlying highly distributed and low-complex infrastructure.

(2) Automatic Orchestration. Application developers need a hassle free way to deploy, test, and update their multitenant applications to remote nodes in a seamless manner and ensure the services are running smoothly with high availability. However, at the present time applications spanning cloud and edge are still provisioned manually. Different from current cloud infrastructure in data centers, city-wide infrastructure contains thousands of geo-distributed wireless edge nodes deployed in residential areas, streets, or parks. Enabling automatic application orchestration in each edge node with high reliability is unachievable using existing solutions. To provide a robust environment where developers can remotely deploy and debug their applications in the cloud of edge nodes, the orchestration layer with the remote management dashboard is a must-have to enable automation.

(3) Resilience and Availability. Due to the limited computing and storage resources of the edge node, the fully distributed deployment, and the unpredictable service requests, it is hard for developers to ensure resilient application and services. Different from data center networks where cable/fiber connections are more reliable, edge cloud infrastructure involves distributed heterogeneous gateways and unreliable wireless links. As unpredictable disasters and attacks increase, we need a resilient network design for the edge cloud infrastructure that avoids any single point of failure and keeps all the edge nodes connected to vital services. The deployed smart application should be able to easily reroute via the software-defined infrastructure while at the same time collaborating with nearby peers to achieve improved disaster preparedness and response. The adaptive monitoring software will determine when the hardware is likely to fail, when resources will exceed capacity, or where attacks are happening, and finally, deliver the agility and flexibility needed to support multitenant smart applications.

Abstracting the hardware edge node to multitenant applications can help improve device utilization, lower the system deployment cost, and accelerate the real world testing and deployment of new applications. However, to support multitenant isolated applications on an edge device node, an edge cloud software platform with the following capabilities is needed: (1) providing multiple virtual computing resources to multiple applications; (2) supporting application isolation for secure multitenancy; (3) creating a versatile runtime environment that supports a variety of technologies and devices from different vendors; and (4) providing a remote dashboard of the edge cloud that enables third-party application vendors/users to launch and manage applications remotely. To enable remote orchestration of multitenant IoT applications, numerous attempts have been made to extend the flexibility of the cloud into distributed mini data centers such as cloudlets. Stack4Things extends the OpenStack for IoT applications [14] while Openstack++ provides multitenancy through VM based computing resources [15]. This solution of providing hypervisor-based computing resource to establish multitenancy adds a redundant software abstraction layer, thus creating a nondeterministic application. However, the integration of various server-oriented technologies makes the system heavy and expensive to use in IoT applications. Moreover, these solutions do not emphasize the continuity of services and high availability. Edge nodes are commodity hardware nodes distributed in the harsh environments and running computation for applications with unpredictable demand. There lacks a system to provide continuous multitenancy infrastructure services for application vendors when a device in the cluster or an edge node is overburdened.

In this paper, we propose an architecture for a Platform-as-a-Service (PaaS) to automate multi-tenant IoT and CPS

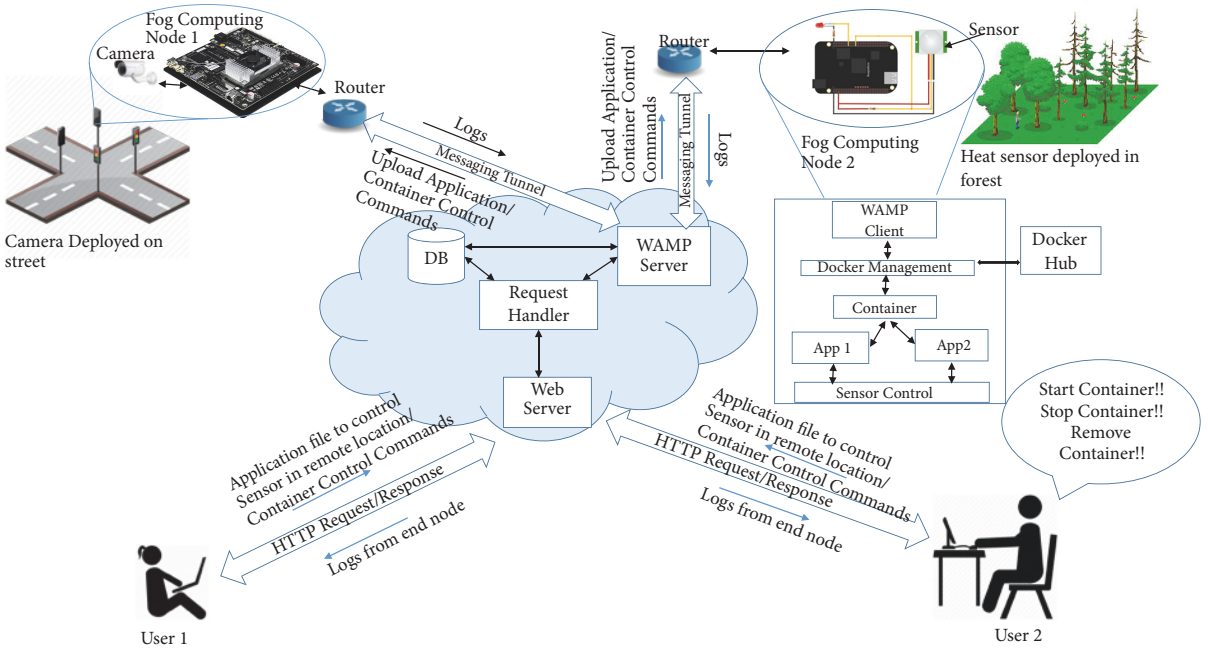


Figure 1: The architecture diagram of system implementation.

applications provisioning in an edge computing environment. Based on our previous results [16], we propose a software framework to enable the envisioned edge cloud platform with three key functionalities: (1) supporting multitenant isolated user applications via lightweight virtualization; (2) providing a cloud dashboard for remote programming and application management; and (3) featuring robust network connectivity agnostic to network mediums. The framework provides computing resources through a lighter, more deterministic container-based virtualization (i.e., IaaS). The computing resource holds all the dependent libraries and runtime environments required to run the isolated application (PaaS). The edge cloud provides a solution offering remote dynamic discovery that enables the user to control and manage applications remotely through a web application dashboard. To ensure service availability and resilience, we propose solutions to seamlessly migrate services amongst nearby edge nodes via lightweight communication protocols when a failure happens or when the resources are overburdened. As a use case, the proposed PaaS was employed to provision multitenant stateless IoT applications that run on top of low complexity edge devices, for example, Raspberry Pi, NVIDIA JETSON, and Beaglebone.

2. System Overview

2.1. System Design. Cloud computing is a promising paradigm with many inherent advantages, such as the easy and efficient provisioning of new applications and resources, not to mention scalability [17]. It encompasses Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS). Application providers use platforms (offered as PaaS) to provision (i.e., develop, deploy, and manage) applications. Edge environment is composed of heterogeneous mobile and low-end devices. The computing capabilities are not comparable to the state of the art servers at the data centers. Existing PaaSs (e.g., Cloud Foundry and Google Cloud Platform) do not enable the provisioning of applications with components spanning cloud and the edge node. Most IoT and CPS applications are manually provisioned today.

We propose a PaaS for these remote edge nodes and organize them as a pool of resources for IoT and CPS applications with collaboration mechanisms to provide high availability and fault tolerance for application developers. To meet these requirements, we focus on two design directions for this edge computing platform. The first is to build a scalable and lightweight execution environment that enables flexible provisioning of resources for multitenant applications over a wide area network (WAN). The second is to provide users a programmable system to manipulate their applications deployed on remote resources and ensure resilience and zero downtime via fault tolerant solutions.

2.2. Overview of System Implementation. (1) Cloud Dashboard. When many applications are deployed using the same IoT infrastructure, resource isolation across different components and between applications is necessary. Container technology is an ideal choice due to its lightweight virtualization and small memory footprint [18]. Since all applications share the OS, restarting a container does not require the OS to restart. Docker's union file system combines layers into a single image, making it ideal for hosting applications in the edge computing platform.

As shown in Figure 1, the cloud dashboard contains three major parts: user interface, database, and messaging services. These three components are interfaced by the request handler module. The requests generated by users in the user interface

are queued by the handler module and then stored to the database to maintain a copy.

User Interface. The user interface provides a user-friendly web application which allows the user to easily create, deploy, start, stop, and remove remote applications. The dashboard serves as the centralized interface for the users to interact with their own applications. For example, users can check the status of their application, control the settings, collect the results, and debug the software. The dashboard can display the collected data as a graph, monitor the computational resources, aggregate the result files to download, and display the available end edge nodes. For example, an authenticated user can login to the dashboard and access his/her edge node. A user can also deploy his/her application to the preferred edge node through this dashboard.

Messaging Service. Messaging services have become an important portion of the Internet of Things. Messaging services not only include communication between two nodes, but also provide security and reliability. The messaging service takes care of transmitting the message to the desired node through the public Internet for the services of device or end node registration, heartbeat information, message transfer, and application deployment. Device or end node registration with the server is also a criterion for our project. The communication between the end node and server, remote configurations to register the end nodes to the server, and the secure communication are fulfilled with the help of our proposed messaging service.

We utilize the Web Application Messaging Protocol (WAMP), which is an open standard WebSocket protocol, to provide the application interface over the *fusion link*. Through the combination of the WAMP with the *fusion link*, sensor devices without internet connection can be reached, which also provides resilient connection services robust to the Internet loss. We support two types of logic connection: remote procedure calls and publish-subscribe messaging services. WAMP is mainly used for distributed, multiclient (edge nodes in our use case scenario), and server applications. Autobahn framework and Crossbar are used to host a server for message routing. Autobahn is an open-source implementation of WAMP and Websocket. Crossbar is a networking platform designed to handle routing for distributed and micro-service applications. The server and client are coupled via a publish & subscribe (PubSub) approach through a router to publish an "abstract" topic to the subscribed client. After the transmission is complete, the server and client are decoupled by the router (also named broker), which keeps track of subscriptions. When an "event" is published by a publisher to a topic, the broker looks up the record to determine the list of subscribers and then forwards the information (the "event") to those subscribers.

Database. The database will be used to save information related to the user, container, the location of the stored results, and so on. We utilize a relational database to establish the relationship between different management entities. To achieve modularity, the database is divided into five components, i.e., user, computation, image, device, and storage. A relationship is established between the modular DB tables to link the user to the resources allocated to him/her on the edge cloud platform. The database access on the cloud is done through a database service which links the DB to other modules through REST APIs. This offers portability and provides language/platform independent interface while offering a loose coupling with other service modules.

(2) Edge Node: **Container Management**. Containers provide a complete software package required to run a user's applications on the edge node by abstracting the underlying operating system. There are different techniques to manage the containers deployed on a machine such as Swarm [18], Kubernetes, Fleet, and so on [19]. Swarm container management is effective and lightweight when compared to other techniques. It converts all the Docker hosts into one single virtual host. All the nodes created on the edge device are controlled, scheduled, monitored, and cleared using swarm manager. Swarm manager also helps us to keep a check on resource utilization.

In our system, containers are launched with applications requested by each user. All dependent libraries to run the application inside a container are installed before executing an application. Every user can deploy a new container for each application and obtain the results. Once the execution of an application is completed, containers are deleted automatically by the Docker daemon.

3. Cloud Orchestration Framework for the Remote Edge Node

3.1. Resilient Network with High Mobility. Existing cloud solutions require either a private network or fixed IP address, which is not achievable for the distributed edge nodes. Most of the edge nodes are connected to the public Internet without fixed IP and some of them even connect through IEEE 802.15-based sensor networks, for example, the 6LoWPAN network. We propose building a *fusion link* solution to manage the distributed edge nodes through lightweight messaging services. We propose utilizing a resilient messaging tunnel to manage the distributed edge nodes, making it agnostic to network mediums as well as supporting dynamic access and mobility support. This messaging service will enable us to contact any remote node based on ID instead of fixed IP. We propose an architectural solution to remotely access edge cloud management services through intermittent Internet connections and close the connection to save power when necessary. Specifically, we build a network overlay above the WAN and 6LoWPAN. When the edge node is not accessible via the WAN, our network overlay will automatically switch the connection from WAN to the 6LoWPAN. Our network overlay also helps to bridge the WAN connection to the 6LoWPAN network, which served as a gateway that seamlessly bridge the Internet world (WAN side) to the sensor world (6LoWPAN side).

Handling Registration and Heartbeat. Every end node which connects to the server is registered with a unique ID. Each unique ID is part of the topic while subscribing. When a user requests to perform actions on the end device, this unique ID is used to publish the request to the desired end node. The periodic heartbeat message is transmitted to the server from all the registered end nodes to make sure the

end device is available for the user. The heartbeat message contains information about the device ID. The very first heartbeat is considered for device registration. The heartbeat consists of device ID, location, and architecture of the device, which are updated to the database upon registration.

3.2. Cloud Dashboard. In the cloud dashboard, user registration is the first step, upon which the user can log in to the user dashboard to orchestrate the end nodes. Various commands such as start, stop, create, and remove a container, as well as upload and download a file from a virtual space on the end node are provided for the user. The requests from the user are received through a REST API. The requests which are intended for the end device are filtered. The filtered requests are then checked for validity, to make sure all the required fields exist in the request. With any field missing, an error message to the user is transmitted. With this validity check on the server, the failure of command execution on the end device is reduced and also the response time to update the error to the user is decreased. Once the validity check is passed, URI is formed based on the device ID and the request raised. Since device ID would be unique for all the registered devices, the "event" is published to the specific device as intended by the user.

Create Container API. Lightweight virtual spaces are provided to users to deploy and execute applications. These virtual spaces can be created by the click of a button from a user dashboard. Each container is given a unique name by appending the username along with the requested container name. During the creation of a new container, if the requested image is not preinstalled on the device, then the image is fetched from the Docker hub and created. Any custom images can be uploaded to the Docker hub by the user requested to pull when required.

Start, Stop, and Remove Container API. Each virtual space can be orchestrated from the user dashboard. Some commands provided to the user are to start, stop, and remove. Each of the commands is transferred to end through the server from the user dashboard. Upon receiving the command, the corresponding API is executed. A negative response is transferred back if the API fails. If not, then a positive response is sent.

Device ID Generation. When the end node application is started, the device name and MAC address of the device are fetched. The same is concatenated in the form "device name / MAC address." With the help of both the device name and MAC address, it is possible to keep each device ID unique, thus helping to differentiate the message to be published according to the device selected by the user.

Location of Device. The location of the device is mapped during deployment and stored in the cloud as one of the key traits of the edge node. This helps to classify each device based on the location. Developers can make use of this feature to deploy their application to a specific location. For example, smart city-based applications utilize the location as the service segment.

Architecture of the Device. The architecture of the end node is fetched and updated during the device registration. We support multiple devices with Linux operating systems, for example, ARM and x86 CPU. Example device architecture includes device name, CPU type, memory, and disk. Container compatibility is based on the architecture. Hence, by detecting the architecture, suitable containers can be listed to the user for deployment. On successful login, a user can access his/her dashboard. In the dashboard, the user can find a list of containers created by all the devices. The user can also control the containers by clicking the container control buttons and see the device information by selecting the device.

4. Edge Node Virtualization

4.1. Edge Node. (1) Device Registration and Maintenance. An edge node can be any embedded device such as Beaglebone Black, Raspberry Pi, Jetson TX1, and so on. Edge node, upon booting up registers to the server with a device name, transmits periodic *heartbeats* to indicate the node is alive. Once the device boots up, a *registration* message is transmitted to the server. The registration message consists of device ID. The device ID is unique to each device. The device subscribes to all the "events" with a unique URI containing the device ID. After registering, the device sends a periodic heartbeat. The *heartbeat* is a short command message to the server to test the connectivity. Once the device boots up, the device checks for an Internet connection and tries reconnecting to the Internet. Once the Internet connection is established, the device forms a device name, detects its location and architecture, and transmits periodically to the server to indicate the end node is alive. The heartbeat stops when the device turns off. When the server did not receive the *heartbeat* for a long period of time, the cloud framework will inform users about the unavailability of the device, hence unregistering from the list for usage.

The node also transmits device information and container status periodically to the server. The device info contains the device ID, device name, memory consumption, CPU usage, disk memory available, operating system on edge node, and kernel version. All this information is contained in the *heartbeat* message and updated to the server periodically. The CPU information can be used to detect the available system resources before deploying containers.

Docker engine is installed on the edge node, allowing the users to deploy their application by creating containers. Docker provides numerous amounts of packages for developers to design and develop application programming interfaces for creating, deleting, running applications, and so on. The edge node communicates with a WAMP agent to receive the commands issued by users. The response from the edge node is sent back to notify the user about the action taken by the Docker. Every application creates a container which is isolated from other containers running on the same Docker engine. This isolation is provided by the Docker technology. Once the Docker engine is installed on the end node, the Docker daemon runs on boot up of the end node and continues to run as long as the end node is not turned off. As soon as a command is issued from the user, the Docker receives a command from the WAMP agent and performs the necessary action. The status of the container is updated

periodically. The status message contains the container name, status, and device name to detect both the device maintaining the container and the image name. Container status can be redirected to inform the user about the status of each container. All status messages are updated to the database and will notify the user if some of the parameters exceed the specified boundaries.

(2) Handle Remote Request and Deploy Application. The subscribed "event" of the edge node receives the request from the server, containing information about the container control. The received request is then parsed to the format as required by the Docker API. During "create" request, each container is renamed according to the user's demand. Container names received through the request are further formatted to fit in the form "username-containerName." By doing so, each container created has a unique name. Once the container is created, a unique container ID returned by the Docker API is tagged along with the container name for more clarity. The response returned by the Docker API is transmitted to the server to indicate the user and also the same in a database.

The application file from the developer can be downloaded from the cloud dashboard to the specific containers in the remote edge node when the user requests. The user can upload their application file via.tar format and command script about running the application. The cloud dashboard will handle all the requests and validate the files through defined policy. User requests, after validated on the server, are transmitted to the respective edge node. The received commands contain the location of application file on the server which is used for download. When the node receives the application file, it extracts the.tar file and runs the command script to execute the application. This process makes it easy and flexible to deploy the application to the remote edge node without being burdened by different run-time environments and configurations during the deployment process. Any applications, as long as they can be formatted as a.tar file and command script, can be deployed through our edge computing dashboard.

5. Ensure Service Resilience via Container Migration

Cyber-physical systems are deployed in a variety of hostile environments that are subjected to changes such as hardware failures, extreme weather, natural disasters, terrorist attacks, and even cyber attacks. The quality of the underlying network that connects these edge nodes varies at different conditions with high dynamics. Lossy network leads to loss of considerable resources and data by genuine users. A resilient and scalable infrastructure is needed to enable the proper operation of remote edge and IoT nodes in this type of remote, hostile environment.

When physical edge nodes are down, we utilize a failover approach using the high-availability images to restore the edge node operating system. However, it takes a long time for this process to restore the whole operating system. We also monitor each container in the cloud dashboard by using *heartbeat* and restoring a container when it is down. However, the data may be destroyed depending on the timing of failures because restoration methods of failed edge nodes and containers are independent. Moreover, with the decrease in size and computational powers of edge nodes, a single node cannot handle the dynamic workload for IoT applications. IoT applications provide sensing and actuating approaches for real-time events. However, demand for the events is hard to predict. Different from data centers that can serve the dynamic demand via pool resources, the local resources available in one edge node are not sufficient for high application demand. An efficient mechanism to migrate services amongst the edge nodes is needed without disturbing the underlying sensor mesh when failure happens or the resources are overburdened. Service migrations between the edge nodes are also important for mission critical IoT operations such as intelligent transport systems or connected vehicles.

Therefore, we propose a fast and reliable restoration method with a uniform method for plural-type virtual resources. In our method, the local framework in the primary edge node predicts the node failure and insufficient resources and notifies the virtual resource arrangement scheduler, after which a virtual resource arrangement scheduler queries the cloud for potential nearby edge nodes for collaboration. It determines both the type of peer nodes required to migrate the container and application data from the primary node to the peer node. The virtual resource arrangement scheduler also buffers new sensor data and restores the application in the peer node for data processing without data loss. We implement the proposed method and show its effectiveness regarding peer collaboration through container migration.

5.1. Connections and Collaborations between Peers. Figure 2 shows an overview of the container migration scenario. Considering intermittent WAN connections, the peer node many not be reached even given close physical distance. To improve connectivity resilience, we propose a *network overlay* over the WAN and low complexity 6LoWPAN technologies for peer connections. If the peer node is connectable via the WAN, the Message Queuing Telemetry Transport (MQTT) protocol will be used to initiate the container migration process. If the WAN connection is lost, our network overlay will automatically pick up the lost connection via 6LoWPAN via MQTT protocol. MQTT is a lightweight machine-to-machine protocol with very little implementation footprinting. We utilize the feature whereby MQTT is available in both WAN and 6LoWPAN. The MQTT broker functionalities are integrated at the edge computing embedded platforms which serve as a message exchange channel for container migration.

When a primary node wants to connect to the peer node with the broker, its makes a connection call first followed by a subscribe message. It has a variable list of topic(s) along with QoS value. The QoS value is defined as the level of collaborations provided by the peer node. We define three levels of QoS values: (1) Level "0": the peer node can provide messaging services and can help contact its own nearby peers for collaboration, but the available resources are not sufficient to migrate the container; (2) Level "1": persistent storage is available to ensure container delivery, but the

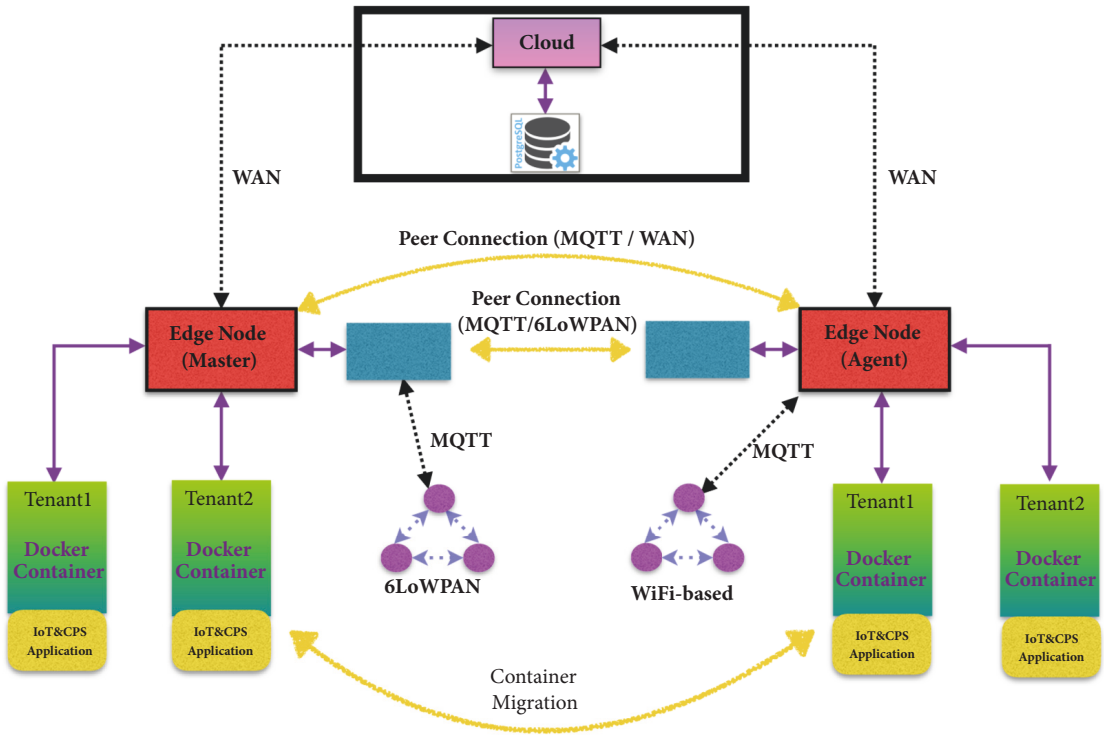


Figure 2: The overview page of the peer-based container migration.

CPU and memory resources are not sufficient for executing the application; and (3) Level "2": persistent storage, CPU, and memory resources are all available for the container migration. It is the highest availability of services, and the primary node can start the container migration immediately.

The message flow of the QoS Level "2" is shown in Figure 3. A message is deleted from persistent storage after publishing a topic only when a PUBREC message from the server is acknowledged with a PUBREL and PUBCOMP. This signifies that the initial sender exchange is completed. Next, the broker sends the message to each subscriber by sending them PUBLISH messages with the message payload. After message receipt, each subscriber sends a PUBREC message in acknowledgment, which the server responds to with a PUBREL (publish release message). After each subscriber acknowledges with a PUBCOMP message, the server deletes the message from the persisted store, and the QoS Level 2 message send sequence is complete.

We also provide the application programming interface for developers to control the process. The available programming interface is shown in Table 1.

5.2. Service Optimization for the Migration. The container migration provides an effective way for us to continue services when the primary node suffers resource overburden or failures. However, performing the migration frequently or in unnecessary conditions lowers the overall performance due to migration overhead. One specific example is the prolonged

Table 1: Application programming interface.

1	getMessage(fileSrc)	Send a file/string
2	connect(deviceID,topic)	establishes a blocking call to the broker
3	sendFile	Send file after getMessage()
4	recieveQ	Retrieve everything from queue
5	receiveSingle	one element at a time from the queue
6	Disconnect()	To disconnect from the broker

latency when the closest edge node migrates the services to the peer node with a longer latency. To optimize the service migration process, we perform quantified decision process for the migration and minimize the overall cost.

Let us assume that the MEC system is composed of a set of N edge nodes (ENs), named $n \sim \Gamma_N, 1 < n < N$, where each EN is referred to with an index of n. We define the number of containers as $c \sim \Gamma_C, 1 < c < C$, where one container or multiple containers belong to one user. We assume one container has one application running for the application isolation purpose. Our system is designed for delay-sensitive applications. This requires the container should be deployed in the nearest EN, to guarantee the desired ultra-short latency t_c^{min}.

Let $d_n(c)$ denote the deployment of container to the edge node n for the c-th container. Assume $T()$ is the latency

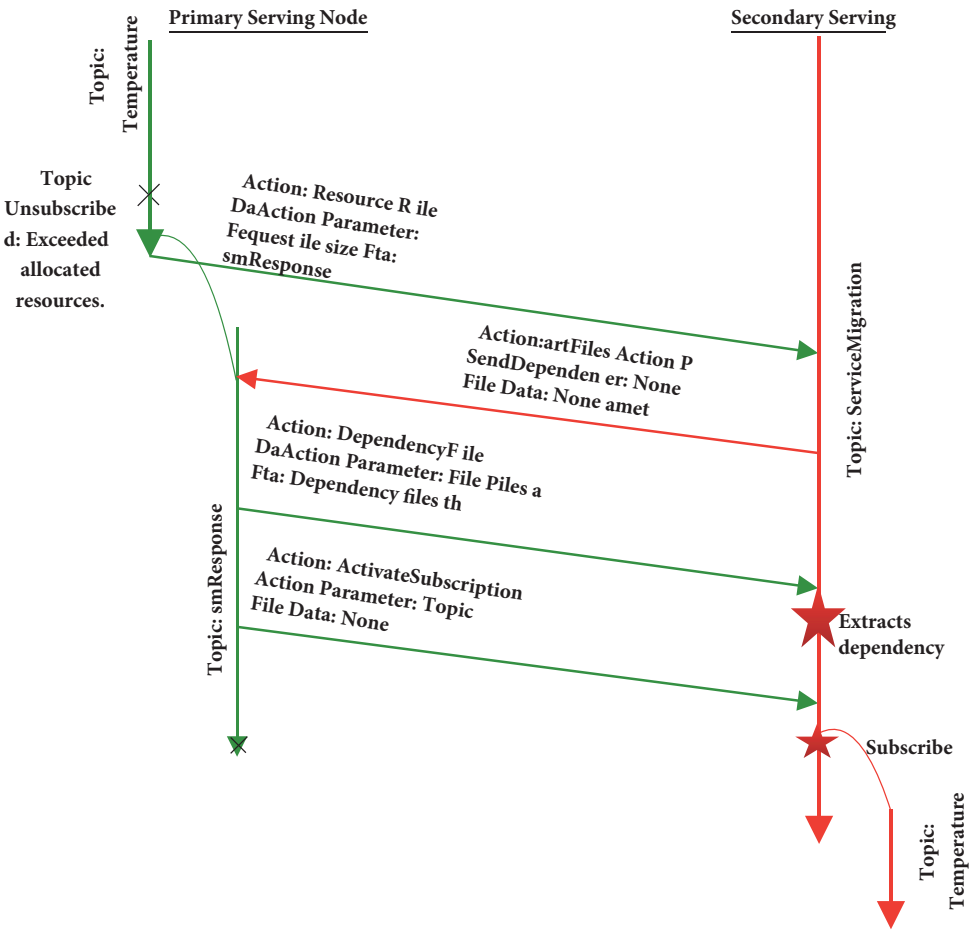


Figure 3: Message flow for QoS Level 2.

estimation function; $U()$ is the computational requirement function; M_n denotes the total computational capability for node n. In the initial step of container deployment, we achieve $T(d_n(c)) = t_c^{min}$, i.e., by deploying the application container to achieve minimum latency. Let $p_n(m)$ denote the probability of peer node selection from node n to node m. The problem of minimizing the overall cost can be formulated according to the following linear program model:

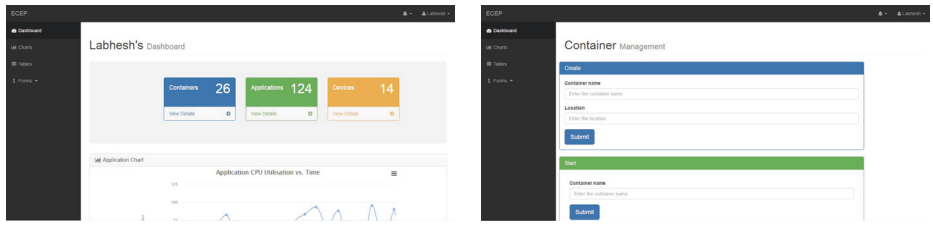
$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \sum_n \frac{T\left(d_n(c)\right)}{M_n - U\left(d_n(c)\right)} \\ \text{subject to} \quad & \sum \left(U\left(d_n(c)\right)\right) < M_n, \quad n = 1, \ldots, N. \\ & t_{th} > T\left(d_n(c)\right) \geq t_c^{min}, \quad c = 1, \ldots, C. \end{aligned} \tag{1}$$

The objective aims to minimize the total cost, which is proportional to the latency and inversely proportional to the remaining computational capability. The constraints account for the limited resources of ENs and the latency test with regard to the threshold t_{th}.

5.3. Launching Docker Image in the Peer Node. Once the peer node has been selected, information about the image, repository, and port is forwarded in the message. This information is extracted at the remote node and a "run" command is performed. The programmer can even choose to send the Dockerfile and its dependencies. The program converts it into a message bean, serialize the sender's end, and deserializes the receivers' end. The image can be built, pushed, and executed. The metadata is sent back to the primary node, which later pushes to the cloud dashboard. For container migration, the image can be check-pointed and restored on another free node.

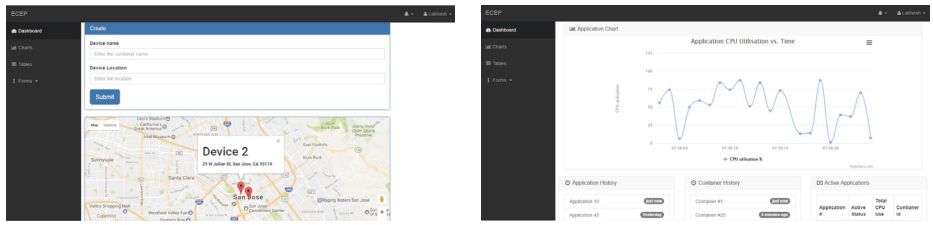
6. Evaluation

Performance is evaluated by deploying the cloud dashboard server on AWS and considering an embedded board with ARM CPU as the end node. The following performance is impacted by the available network upload and download bandwidth. In our analysis, the upload bandwidth is observed to be 11.45 Mbps and download bandwidth is observed to be 24.2 Mbps. To emulate real application data, we utilize one example sensor processing workload by processing the sensor data sent to the edge node for floating point operations. We utilize a 50 by 50 matrix storing sensor data and then perform window-based convolution on the edge node. A total of 5000 floating point multiplications and 2500 floating point



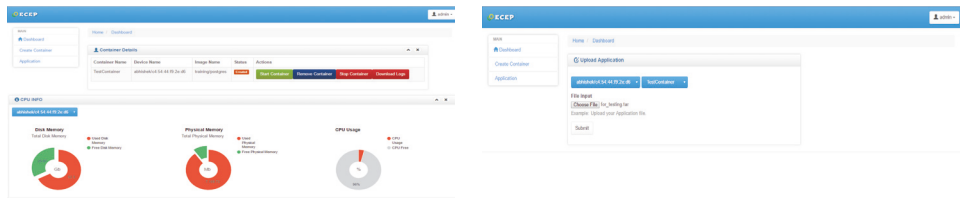
(a) Dashboard (b) Launch containers

Figure 4: The interface of the cloud dashboard and launching containers.



(a) Devices overview (b) Device monitoring

Figure 5: (a) Overview page of the connected devices and (b) the interface of device monitoring results.



(a) Container details (b) Upload applications

Figure 6: (a) The overview of container details; (b) the interface of uploading applications to the container.

additions are performed with every set of data sent by the sensor.

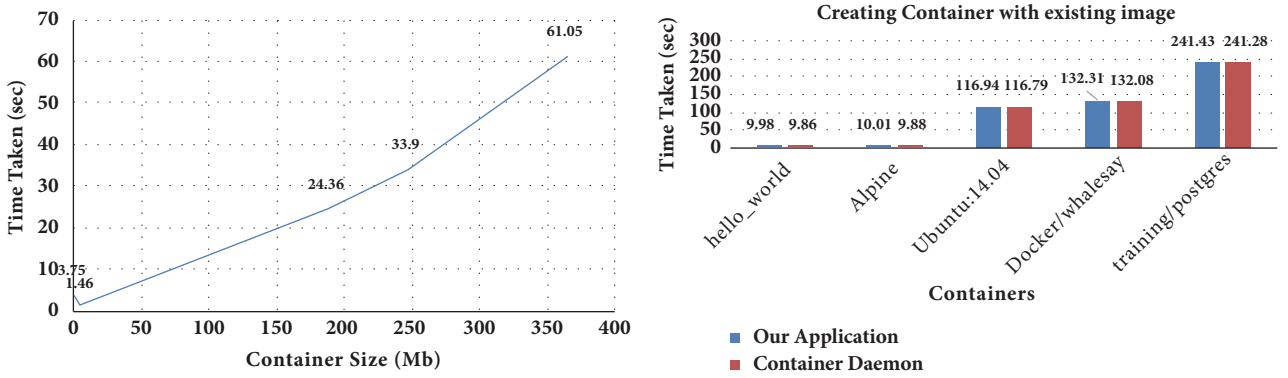
6.1. Cloud Dashboard. Figure 4(a) shows the interface of the cloud dashboard. This interface provides an overview for developers to control the infrastructure. The available containers and devices are listed in this overview. Figure 4(b) shows the interface for developers to launch containers to the remote edge node.

Figure 5(a) shows the list of available edge nodes in the cloud dashboard. When the developer wants to get the detailed information of the edge node, he or she can click it and get the overall device monitoring results as shown in Figure 5(b).

When one developer wants to deploy the application to one container in the edge node, he or she can view the details of the container as shown in Figure 6(a) and upload applications by selecting the files as shown in Figure 6(b).

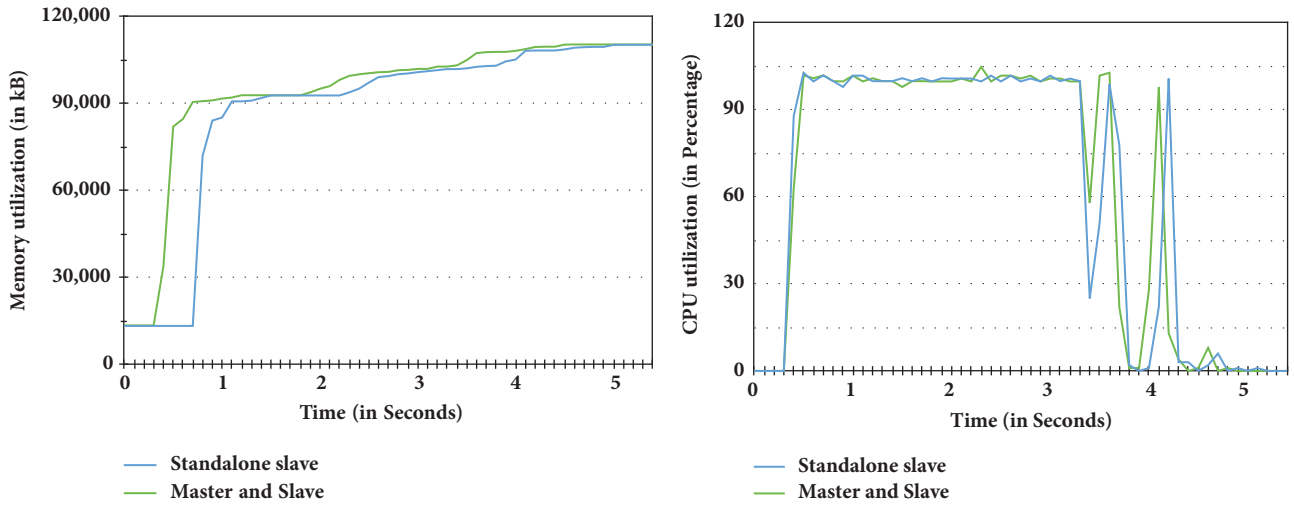
6.2. Launch and Manage Containers from the Cloud. **Performance for Creating a Remote Container by Downloading a New Image from the Cloud**. Figure 7(a) shows the performance for creating a remote container by downloading a new image from the cloud. This is the first step when users want to deploy their applications to the remote edge node. Users can select the type of the container image and push it to the edge node. Figure 7(a) represents the time required to pull an image from Docker hub in the cloud and create a container in the edge node from the downloaded image. For performance analysis purpose, we consider different five images with sizes ranging from 1.5kB to 350MB. The calculated time includes the download time of the images and the time for creating a container out of them. Network latency is assumed to be equal during the course of evaluation.

It takes approximately 3.75 seconds to pull a hello-world image (which is of 1.8kB in size) from the Docker hub and create a container. Upon issuing the command, the Docker daemon looks for the image in the cache, and if it does not find it, then it will pull the latest image from the hub and creates a container using this image. Similarly, the "training/postgres" image consumes a time of just over a minute to pull and deploy a container. When compared with the small size image, there is an exponential increase in the time factor. The reason behind this is that the Docker daemon requires more time to create an image for large packages in addition to the linear downloading time. Figure 7(a)



(a) New image (b) Existing image

FIGURE 7: (a) The time taken to create containers for new images with different sizes and (b) the time taken to create containers for existing images with different sizes.



(a) Memory (b) CPU

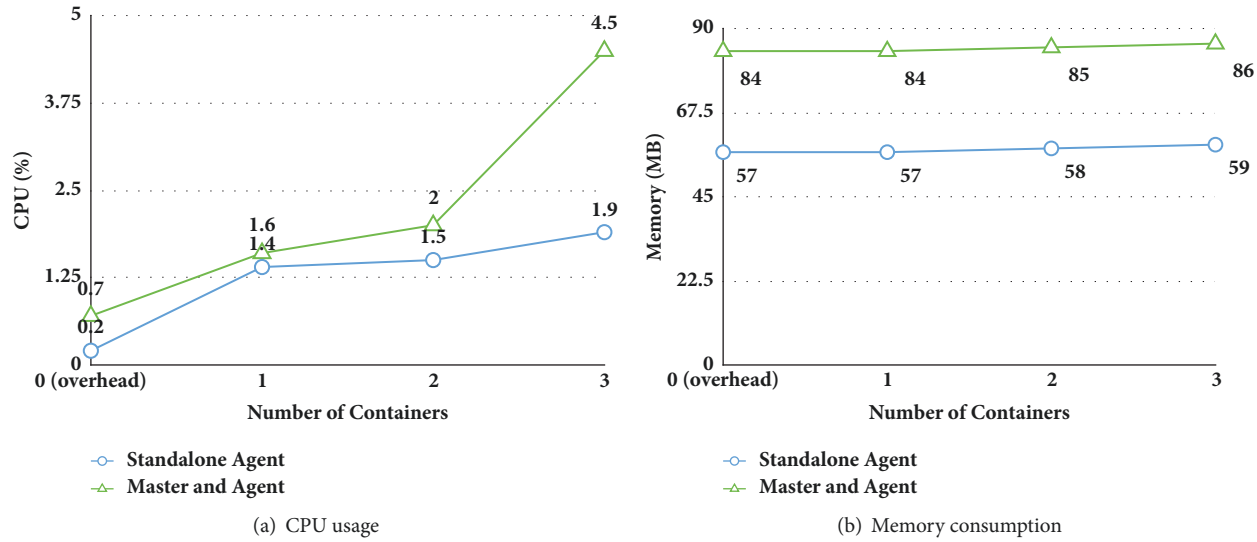
FIGURE 8: (a) Memory utilization of framework and (b) CPU utilization of framework.

clearly suggests that bigger image sizes require more overhead time, including downloading and creating a container of that image.

Performance for Creating a Remote Container via Local Existing Images. Figure 7(b) represents the time required by the Docker daemon to create a container using an image stored in the cache. When a user requests to create a container with a specific image, the application passes the image name and the container name to the Docker daemon. Docker daemon then looks through the cache to find out if the image is stored in it or not. If the image is available, it will pull the existing image from Docker hub and create the container. The performance measured here is the comparison between the time taken to create a docker container with the help of docker and the time taken to process all the commands received from the user.

To evaluate the performance, we downloaded 5 images with various sizes and applications. Our service layer consumed approximately 10 milliseconds to create a container with the "hello-world" image. Similarly, 241 milliseconds is required to create a container using the "training/postgres" image. The reason behind the time difference is that the latter creates a database in a container which consumes more time when compared with the earlier one which only prints "hello-world."

6.3. Multitenant Applications for the Edge Node. **Resource Utilization of the Docker Image**. To estimate the resource utilization of the docker image in practical IoT scenarios, we launched the Master and Agent on two edge nodes. From Figure 8(a), the memory utilization of the Master (master and slave services are all running) is slightly higher than the standalone slave. From Figure 8(b), we observed that our framework's CPU utilization remains high when it launches its services and spikes when scheduling the container. After the containers are fully scheduled, minimal overhead is observed. While the tasks are running, the node can dedicate all the system's resources for the IoT application needs.



(a) CPU usage

(b) Memory consumption

Figure 9: (a) CPU usage versus the number of launched containers and (b) memory consumption versus the number of launched containers.

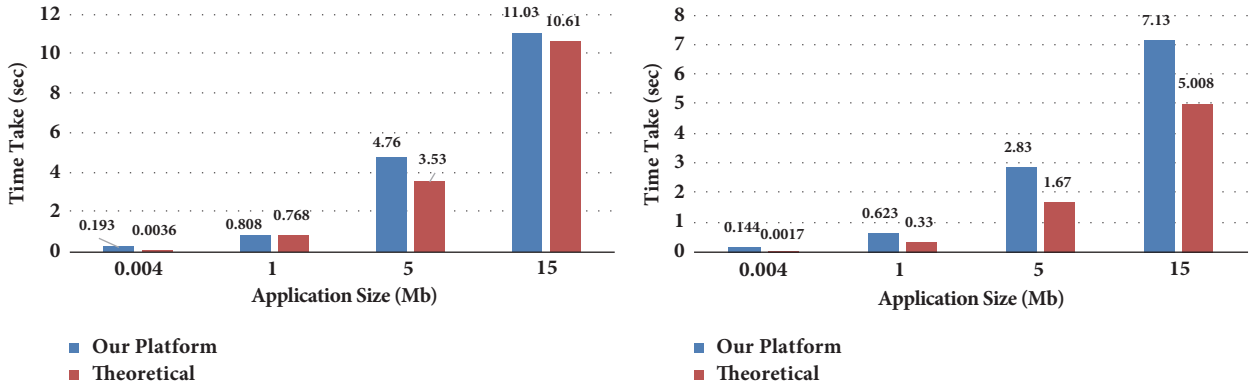
Framework Overhead. To test the case of the framework overhead for supporting multitenant applications, we evaluated the CPU and memory consumption of the edge node when multiple containers are launched on the two agents. As shown in Figures 9(a) and 9(b), the CPU time and memory are calculated by averaging the results from ten attempts of launching multiple containers varying from 0 ~ 3. All in all, CPU consumption of the standalone agent is 2.36 times better than the master, while the overall CPU overhead is small, with less than 5%. The memory overhead does not change significantly when increasing the number of containers since the tenant container resides on the disk rather than the memory.

Hardware Independence. The total time required to create the container depends on the image size and type as well as the overhead time of our service layers in our edge computing framework. We developed the service layer using messaging-driven transparent APIs and make it independent of different hardware platforms. This flexibility is very important in utilizing different hardware platforms from different vendors to create a common edge-computing platform for multitenant applications. To analyze the system overhead, we performed the container creation operation on different platforms, i.e., Beaglebone Black, NVIDIA Jetson TX1, and Intel desktop computer having the configuration of Intel i5-3320M, a clock speed of 2.6GHz, and 8 GB of physical memory. It was observed that container creation time does not differ much from different platforms. The difference of about 0.15 milliseconds is the amount of time consumed by our platform on the end node. This time is approximately equal across platforms: Beaglebone Black (ARM CPU), Jetson TX1 (ARM CPU), and the computer with an Intel CPU (X86).

Container Density. The key feature of our proposed edge-computing platform lies in the multitenancy support. Specifically, one edge node can launch multiple isolated containers and support multiple developer groups simultaneously without interference. The number of containers to be supported on an edge device depends on the available computing resources and the size of each container. We tested the following platforms: (1) the Beaglebone Black with total disk space of 4GB, available disk space of 1.8GB; (2) NVIDIA Jetson TX1 with total disk space of 16GB, available disk space of 4.6GB; and (3) Intel computer with total disk space of 256GB, available disk space of 118GB. One Ubuntu image of 188MB was considered for analysis. On Beaglebone Black, we observed that, with the creation of eight containers, the system was overloaded and slowed down. Hence, it is recommended to use light weight containers with the size of 5MB to 40MB. On Jetson TX1, 15 containers were created and the system performance did not degrade. There were no limitations on the computer since the available disk space was much higher.

6.4. Deploy and Manage Multitenant Applications. Our dashboard provides functions for users to upload their applications to the remote containers in the edge node. The applications can be the source code, compiled executable files, or Jave compressed packages. To facilitate multiple application file requirement, we utilize a common format (.tar) for all applications. When users want to upload their applications to the edge node, they need to prepare their application in terms of the.tar format. Then, the web server will transfer the application file uploaded by the user to the edge node through our messaging service layer using two steps: (1) transfer the application file to the web server and (2) transfer the application file from the web server to the edge node. The file size varies from 4 KB to 15 MB. The performance analyzed here is in comparison to theoretical values calculated by the formula "file size × 8 / network bandwidth." Overhead of 50% is added for file sizes lesser than 1 MB and 10% for file sizes greater than 1MB.

Transfer the Application File to the Web Server. To evaluate the performance of uploading an application file



(a) To the web server

(b) To the edge node

FIGURE 10: The time taken to transfer the application file (a) to the web server and (b) to the edge node.

from the user dashboard to WebServer, we perform time consumption calculations based on a fixed network bandwidth (11.45Mbps as the upload speed). The network bandwidth is assumed to be constant over the course of evaluation.

The web server reads the compressed application file using "multer" and uploads serially through an HTTP request. The uploaded file is then moved to a unique storage space with a unique ID. When the file is too big, we divide them into several small compressed files, which are then transferred. The total time consumed includes the file compression, division, and transfer. As shown in Figure 10(a), the results show that the time consumed is almost linear with the increase in application files. The difference in time observed is due to the time taken by our platform to process the data, create a directory, upload the file, and update the database.

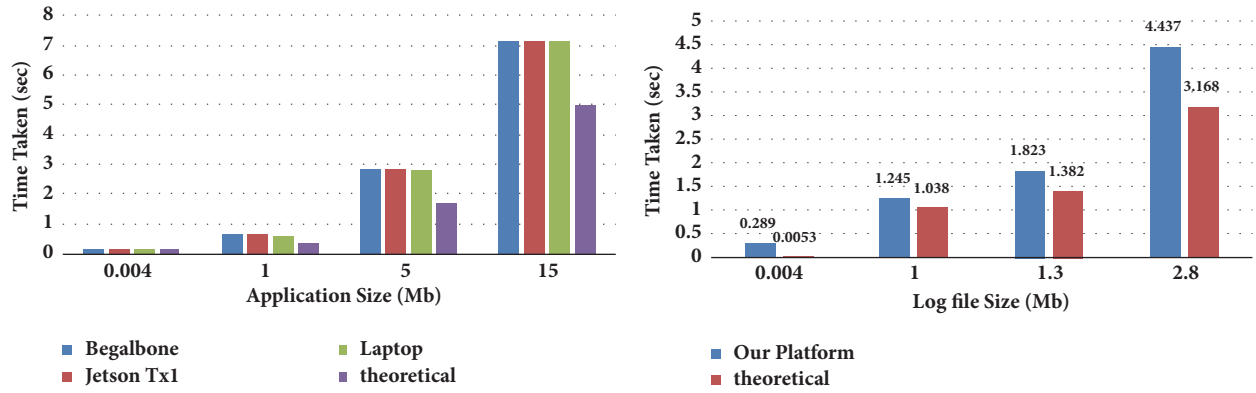
Transferring the Application File to the Edge Node. Application files uploaded by the user are transferred from the user dashboard to the remote node and installed by the application into the deployed container. This section analyzes the performance by measuring the time to transfer the application file from the server to edge node. The fixed download bandwidth (24.2Mbps) is utilized in the evaluation process. The network bandwidth is assumed to be constant over the course of evaluation.

Our platform transfers the compressed application file by reading a chunk of data stored on the server and transmitting it to end node serially through an HTTP request. The received chunk of data is stored on a file locally on the end node. Several compressed files were transferred from server to the edge node, and the time consumed by each compressed file was recorded. The file size varied from 4 KB to 15 MB. As shown in Figure 10(a) and Fig. **??**, the time consumed was almost linear with respect to the application size. The difference in time observed is due to the time taken by our platform to fetch the data, assemble the file, and load the application to the container. This analysis was performed on the hardware of the Begalebone Black.

To test the performance independence on different hardware platforms, we conduct the evaluation of the time taken to transfer the application file to the edge node in different platforms: Begalebone Black, NVIDIA Jetson TX1, and a computer having the configuration of Intel i5-3320M, with 2.6GHz clock and 8GB of physical memory. As shown in Figure 11(a), there was a negligible amount of time difference when performed on different platforms.

Sending Commands and Debugging the Edge Node. The WAMP messaging protocol was used in our platform to communicate between server and end node, for example, transmitting a command from the user dashboard to the end node to obtain a response. WAMP protocol is an asynchronous communication protocol, which is an added advantage of our platform. Our platform makes use of the publish and subscribe feature of WAMP. While transmitting a command from the server to end node or while transmitting a response from the end node to the server, commands and response are noted in a string format. The string size differs for various commands and responses. It was observed that the time taken to transfer a string of 56 characters (56 bytes) was around 1.68 milliseconds. Different commands from users were translated to a string which constituted different sizes; hence time consumed to transfer a command from the server to end node varied from 1.68 milliseconds to 2.37 milliseconds. Similarly, a response from the end node to the server was timed, and it was observed that the time consumed was in the range of 2.53 milliseconds to 2.84 milliseconds.

As a user requests to fetch the log file for the debugging purpose, a command from the cloud dashboard is transferred to the edge (end) node and fetches the log result file from the edge node to the cloud dashboard. The transfer of the file from the edge node to Webserver is achieved with the help of HTTP protocol. The log file available on the end node is read in the form of chunk and is transmitted from the edge node to cloud dashboard. The received chunk is copied onto a file on the server until the end of the file. Different sized result files were considered to measure the time taken. An upload speed of 11.45Mbps is considered during the evaluation. Figure 11(b) shows the required time to fetch the log file with respect to different file sizes. The difference in time is due to the time caused by our platform to send a message from the Webserver to end node, process the request, copy the file from the container to the local path,



(a) Different platforms (b) Log files

FIGURE 11: (a) The time taken to transfer the application file to the edge node in different hardware platforms and (b) the time taken to transfer the log file from the edge node to the cloud dashboard.

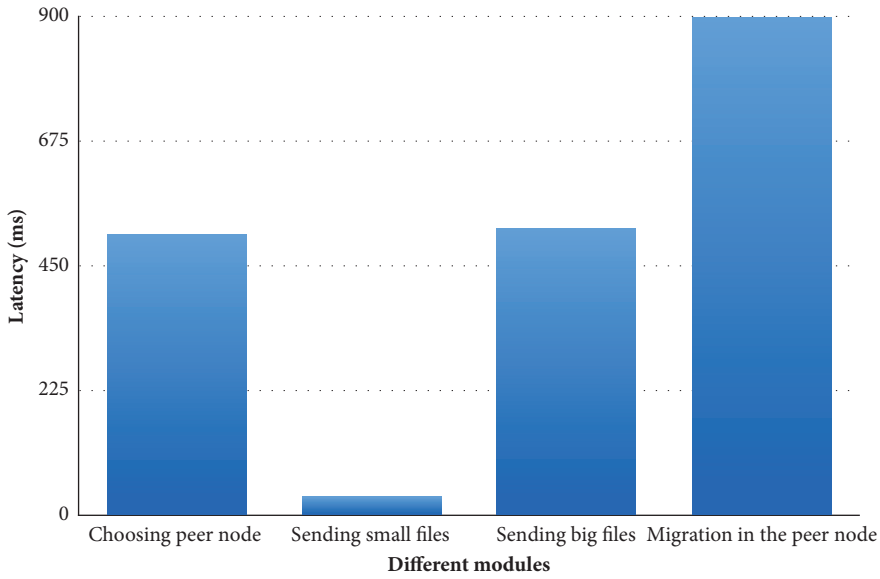


FIGURE 12: The latencies for different components.

locate the file, and then transfer to the cloud. This analysis measures the time required for the complete procedure which consists of sending a command from user dashboard, routing the command to a particular edge node, copying the file from container to the local path on the edge node, transferring the file from to the server, pointing to the messaging service lay to download, and transferring of file to the user dashboard.

6.5. The Evaluation of the Service Migration. **Latency Test of the Service Migration**. To quantify the latency of the service migration between peer nodes, we conducted the latency tests for different modules at the different stages of the service migration. Figure 12 shows the latency costs for different modules in service migration. We evaluate the latency in the primary node and the peer node, respectively. Module 1 (selecting the peer nodes), module 2 (sending small configuration files), and module 3 (sending big dependency files) in Figure 12 are measured in the primary node. The completion of the protocol to choose a working peer node takes 586 ms. Once the working secondary node is chosen, sending configuration and dependency files is much faster. For example, 60kB of data takes 36 ms. The delay of module 4 is measured in the peer node, which calculates the total migration time taken between receiving the "Resource Request" and finishing the "Activate Subscription."

Resource Utilization during Service Migration. This experiment uses two edge nodes as one example to evaluate the coordinated service migration and the impact on the CPU utilization. Once the error happens, or the capacity limits are reached in the primary edge node, the agent in the primary edge node will initiate a service migration request to its peer node. At this point even if connected sensor nodes continue to publish messages, the MQTT broker stores the messages in the queue until the service is migrated. The exchanged messages between these two peer nodes include commands and the corresponding executable routines. When the initial

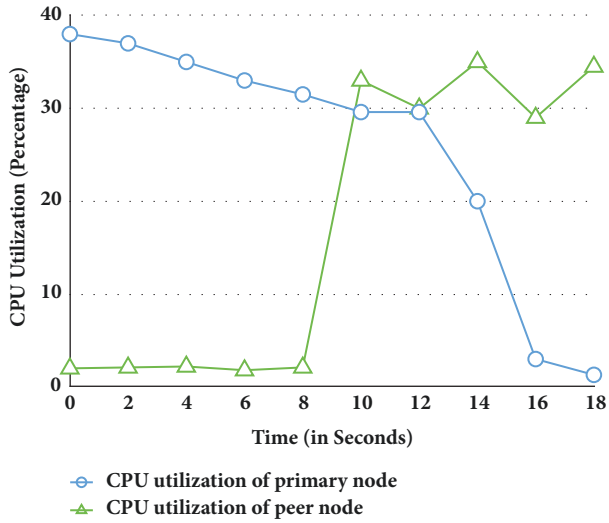


Figure 13: The CPU utilization of the primary node and peer node during service migration.

Table 2: Comparison of our platform with Stack4Things.

Features	Our Solution	Stack4Things(Existing)
Multi-Tenancy	Yes	No
Target-Application	Any	IoT Sensing.
Remote Access	Yes	Yes

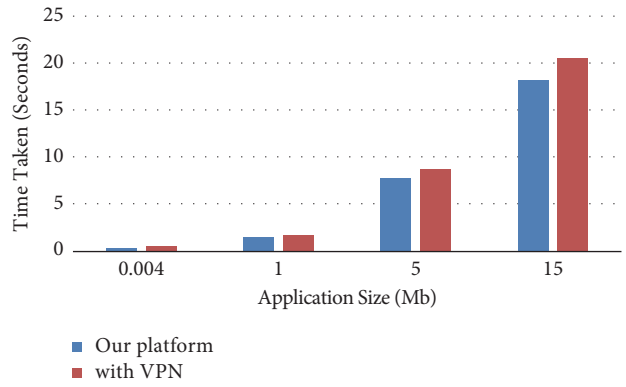


Figure 14: The performance comparison with VPN based approaches.

message exchange is finished, the primary node starts to transfer the dependent files to the peer node. Figure 13 shows the CPU workload of the two peer nodes during migration. When the service migration happens, the CPU utilization of the primary gradually reduces to zero, while the peer node's CPU utilization increases after the user's services are migrated. The peer node starts to perform the service computations received from the data source. The initial spike in the secondary node's CPU performance is due to the processing of the "RequestResource" command.

6.6. Comparison of Our Platform with VPN-Based Solutions. As shown in Table 2, when comparing with existing solutions, for example, Stack4Things [14, 20], the advantage of our solution includes multitenancy support in the remote edge node, compatible with various applications, which allows remote access as well as remote debugging and data transfer.

The second type of related solution is VPN-based cloud solutions [10]. Resin.io is one of the commercialized solutions that utilize the VPN network connecting the distributed Linux containers [21, 22]. They utilize a VPN-based IP network to connect various remote edge nodes (outside the data center) through the public Internet. VPN-based solutions make it simple to apply existing cloud solutions for the remote edge computing platform. A VPN-based remote access mechanism is one of the commonly used approaches to access a remote node on a private network. For example, existing cloud solutions require a private network with a fixed IP address to connect to a peer computing node. Leveraging the VPN connection, the private network can be expanded to the network edge. This works by creating a secured network connection over the public network like the Internet. VPN can connect multiple sites over large distances typical of a WAN (Wide Area Network). Typically, all the users connected to and authenticated by a single public VPN server (which again routes the user requests over the Internet to another VPN server) are distributed in various geographical sites where the edge device is deployed. All the devices deployed are part of a virtualized private network. However, this kind of solution has multiple limitations: it (1) does not support dynamic edge node mobility; (2) has high overhead in terms of connecting remote edge nodes; and (3) is not robust in terms of linking failure.

To compare our approach with VPN-based solutions, we deploy an OpenVPN server in the Amazon EC2 Server. The edge devices connect to the VPN server to be a part of a single private network accessible from the server. A webserver to upload and download a file is run on the server, as well as a client to measure the upload and download throughput. As shown in Figure 14, our approach is 12.6% better than VPN for various application sizes. The reason is that our approach is more lightweight and flexible than VPN-based solutions.

The summary of the differences is included in Table 3. Our major advantages include lightweight connection, support dynamic node access, flexibility in terms of connection failure, and congested bandwidth.

6.7. Comparison of Our Container Migration Solution with CRIU. Checkpoint/Restore In Userspace (CRIU) is a software tool that can freeze a running application (or part of it) and checkpoint it as a collection of files on disk for the Linux operating system. You can then use the files to restore the application and run it exactly as it was during the time of the freeze. Many projects are using this feature for the container migration process. CRUI currently used by OpenVZ, LXC/LXD, Docker, and other software, and CRIU packages is included into many Linux distributions.

To compare the performance of our custom designed container migration scheme, we utilize the freeze time as the performance metric. A smaller value of freeze time means

Table 3: The performance comparison with VPN-based approach.

Our Approach	VPN based remote access
Requires no network configuration. Provides automatic registration of device when deployed in any private network.	Requires configuration of VPN server running parallel with the platform service on the server.
Gives a consistent latency for upload and download, independent of the location of the device.	Latency is highly influenced by the distance between the device and the server.
Gives auto-reconnect when the device is deployed in unreliable network.	VPN server may or may not provide auto-connectivity.
Requires limited bandwidth as it uses WAMP messaging.	VPN connectivity relies on high bandwidth as it channels the network over the Internet.

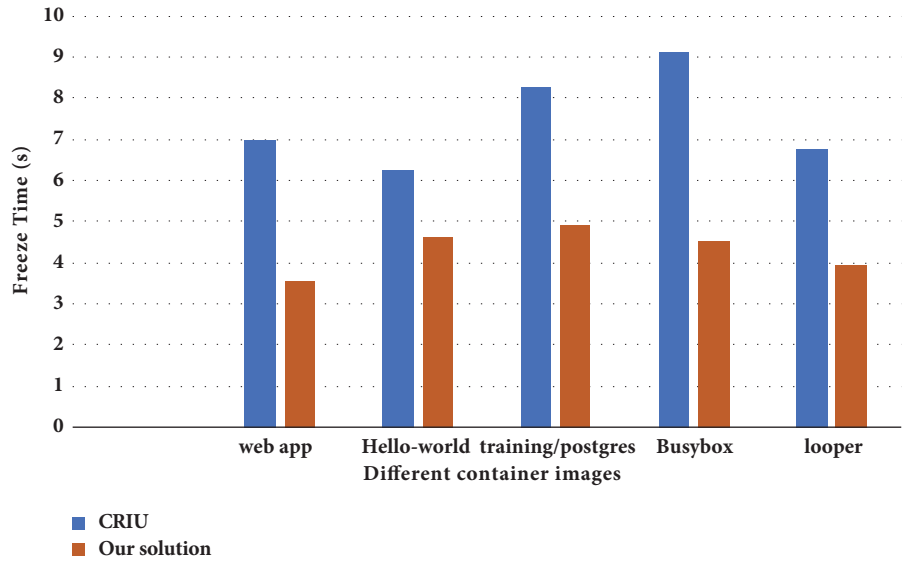


Figure 15: Comparison of our container migration solution with CRIU.

better service availability. As shown in Figure 15, our solution achieves better service availability than CRIU for different container images, i.e., web app, hello-world container image, sql server training/postgres, a busybox in backend, and a looper image which loops numbers indefinitely.

7. Related Work

Mobile Edge Computing (MEC). Mobile edge computing (MEC), which provides cloud computing capabilities, offers a new paradigm to liberate mobile devices from heavy computation workloads [4]. In conventional cloud computing systems, remote public clouds, e.g., Amazon Web Services, Google Cloud Platform, and Microsoft Azure, are leveraged, and thus long latency may be incurred due to data exchange in wide area networks (WANs) [5]. In contrast, MEC has the potential to significantly reduce latency, avoid congestion, and prolong the battery lifetime of edge devices. The first form of the edge computing is the cloudlet [23–25]. The idea behind the cloudlet is to place computers with high computation power at strategic locations in order to provide both computation resources and storage in vicinity [4]. How to offload the computation tasks from the mobile devices to a physically proximate MEC server remains a major research challenge [4, 17, 25].

A more general concept of the edge computing, when compared to cloudlets, is known as a fog computing (introduced by Cisco in 2012) [8]. The key motivation is to enable a processing of the Internet of Things (IoT) and big data applications on billions of connected devices at the edge of network [26]. Cisco, along with other big industry players, e.g., Intel, Dell, and ARM, formed an open fog consortium [11] in 2015. While the MCC is a fully centralized approach with farms of computers usually placed at one or few locations, edge computing is meant to be deployed in a fully distributed manner. On the other hand, the edge computing provides only limited computational and storage resources with respect to the mobile cloud computing (MCC). With fog computing, some of the components of an application could be hosted and executed in a cloud Platform-as-a-Service (PaaS) and interact with the other components hosted and executed in the fog, thus closer to the end-user and/or data sources such as wireless sensors [17].

As edge computing is gaining the much-deserved popularity, recent years have seen the evolution of many edge computing platform solutions. These platform solutions often

provide a set of software services that offloads the data on the network, which is then usually sent to the centralized cloud. Some of the major players in the edge cloud are companies like Cisco and Akamai [10, 27–29]. Akamai cloudlets provide vendor application that is designed to solve specific business and operational challenges [20, 30]. This is Software as a Service at the network edge. Cisco IoX [28] is a fog computing platform that provides a single VM instance of Linux for computing which runs beside their network OS on routers, switches, etc. The application runs alongside the network OS that obtains sensor data from the wireless network for the edge computed tasks. CMU Cloudlets [12] adopt a similar framework of fog computing, in which a Cloudlet server, similar to the Fog server, is deployed in the proximity of mobile users and processes the computing requests of mobile devices in real time for video streaming and data processing.

OpenStack [31, 32] is an open source cloud management software platform that manages the underlying hardware infrastructure to provide computing, storage, and networking resources to third-party user applications. OpenStack can be extended to cloudlet at the network edge to provide robust services [15]. For example, Stack4Things [14] and OpenStack++ [15] provide an OpenStack-based edge cloud framework for "Sensing and Actuation as a Service" [33, 34]. The open edge computing community also claims to provide the extension of OpenStack to the cloudlet [11]. However, the integration of various server-oriented technologies makes the system heavy and expensive to use in low-complex IoT and CPS applications.

8. Conclusions

This paper proposes an edge computing platform solution for developers to remotely orchestrate edge devices without caring about their physical location, or their network configuration. Leveraging the minimal usage of network bandwidth via asynchronous communication between servers and clients, we enable developers to deploy applications in a virtualized space, debug, analyze their performance, and retrieve the results of the remote applications. We utilize the Docker technology to provide a lightweight virtual space in the form of containers, which consume less memory when compared to virtual machines, thus providing an advantage due to the memory constraints of embedded environments. To improve the system resilience, we proposed a *fusion link* that automatically selects the available connectivity services based on WAN and 6LoWPAN. We utilize the MQTT protocol to provide a unified connectivity interface for developers. Our platform is useful for applications that require low latency with a dynamic and unpredictable workload, and also in some applications where there is no fixed IP address for connections to be established between the user and end nodes. For the future research direction, we will further reduce overhead when deploying the applications. Specifically, we will enable a secure shell option to a specific container, which will allow developers to have fine-grained control of the container. We will also enable sensor and hardware interface sharing between multiple containers.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Disclosure

Abhishek Gurudutt is now with Nexteer Automotive. Tejeshwar Kamaal is now with Simplehuman LLC. Chinmayi Divakara is now with Amazon Lab126. Praveen Prabhakaran is now with Owl cameras.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The work presented in this paper is funded by Cisco Systems and National Science Foundation under Grant No. CNS 1637371.

References

[1] I. Stojmenovic, "Fog computing: A cloud to the ground support for smart things and machine-to-machine networks," in *Proceedings of the 2014 Australasian Telecommunication Networks and Applications Conference, ATNAC '14*, pp. 117–122, Australia, 2014.

[2] T. H. Luan, L. Gao, Z. Li, Y. Xiang, G. Wei, and L. Sun, "Fog computing: Focusing on mobile users at the edge," https://arxiv.org/abs/1502.01815, 2015.

[3] M. Yannuzzi, R. Milito, R. Serral-Gracia, D. Montero, and M. Nemirovsky, "Key ingredients in an IoT recipe: fog computing, cloud computing, and more fog computing," in *Proceedings of the IEEE 19th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD '14)*, pp. 325–329, Athens, Greece, 2014.

[4] P. Mach and Z. Becvar, "Mobile Edge Computing: A Survey on Architecture and Computation Offloading," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[5] Y. Mao, J. Zhang, and K. B. Letaief, "Dynamic Computation Offloading for Mobile-Edge Computing with Energy Harvesting Devices," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 12, pp. 3590–3605, 2016.

[6] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: Architecture, applications, and approaches," *Wireless Communications and Mobile Computing*, vol. 13, no. 18, pp. 1587–1611, 2013.

[7] C.-H. Hsu, S. Wang, Y. Zhang, and A. Kobusinska, "Mobile Edge Computing," *Wireless Communications and Mobile Computing*, vol. 2018, Article ID 7291954, 3 pages, 2018.

[8] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, MCC '12*, pp. 13–16, ACM, 2012.

[9] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, "Fog computing: A platform for internet of things and analytics," in *Big Data and Internet of Things: A Roadmap for Smart Environments*, pp. 169–186, Springer, 2014.

[10] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *Computer Communication Review*, vol. 44, no. 5, pp. 27–32, 2014.

[11] G. I. Klas, "Fog computing and mobile edge cloud gain momentum open fog consortium etsi mec and cloudlets," 2015.

[12] M. Satyanarayanan, "Cloudlets: at the leading edge of cloud-mobile convergence," in *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, pp. 1-2, ACM, 2013.

[13] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2795–2808, 2015.

[14] F. Longo, D. Bruneo, S. Distefano, G. Merlino, and A. Puliafito, "Stack4Things: An OpenStack-Based Framework for IoT," in *Proceedings of the 2015 3rd International Conference on Future Internet of Things and Cloud (FiCloud '15)*, pp. 204–211, 2015.

[15] K. Ha and M. Satyanarayanan, *Openstack++ for Cloudlet Deployment*, School of Computer Science Carnegie Mellon University Pittsburgh, 2015.

[16] K. Liu, A. Gurudutt, T. Kamaal, C. Divakara, and P. Prabhakaran, "Edge computing framework for distributed smart applications," in *Proceedings of the 2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, IEEE, 2017.

[17] S. Yangui, P. Ravindran, O. Bibani et al., "A platform as-a-service for hybrid cloud/fog environments," in *Proceedings of the IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN '16)*, pp. 1–7, IEEE, 2016.

[18] S. McDaniel, S. Herbein, and M. Taufer, "A Two-Tiered Approach to I/O Quality of Service in Docker Containers," in *Proceedings of the 2015 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 490-491, IEEE, 2015.

[19] D. Bernstein, "Containers and cloud: from LXC to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[20] G. Merlino, D. Bruneo, S. Distefano, F. Longo, and A. Puliafito, "Stack4Things: Integrating IoT with OpenStack in a Smart City context," in *Proceedings of the 2014 International Conference on Smart Computing Workshops, (SMARTCOMP Workshops '14)*, pp. 21–28, IEEE, 2014.

[21] A. Wilson, "Cellular man-in-the-middle detection with stich," *Linux Journal*, vol. 2017, no. 274, p. 1, 2017.

[22] "Resin.io makes it simple to deploy, update, and maintain code running on remote devices," https://resin.io.

[23] G. A. Lewis, S. Echeverría, S. Simanta, B. Bradshaw, and J. Root, "Cloudlet-based cyber-foraging for mobile systems in resource-constrained edge environments," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 412–415, ACM, 2014.

[24] G. Lewis, S. Echeverria, S. Simanta, B. Bradshaw, and J. Root, "Tactical Cloudlets: Moving Cloud Computing to the Edge," in *Proceedings of the 2014 IEEE Military Communications Conference (MILCOM '14)*, pp. 1440–1446, IEEE, 2014.

[25] K. Ha, Y. Abe, Z. Chen et al., "Adaptive vm handoff across cloudlets," Technical Report CMU-CS-15-113, CMU School of Computer Science, 2015.

[26] J. Zhu, D. S. Chan, M. S. Prabhu, P. Natarajan, H. Hu, and F. Bonomi, "Improving web sites performance using edge servers in fog computing architecture," in *Proceedings of the IEEE 7th International Symposium on Service-Oriented System Engineering (SOSE '13)*, pp. 320–323, IEEE, 2013.

[27] A. V. Dastjerdi, H. Gupta, R. N. Calheiros, S. K. Ghosh, and R. Buyya, "Fog computing: Principles, architectures, and applications," https://arxiv.org/abs/1601.02752, 2016.

[28] R. Mora, "Cisco iox: Making fog real for iot, blogs@ cisco-cisco blogs," 2015.

[29] Z. Pang, L. Sun, Z. Wang, E. Tian, and S. Yang, "A Survey of Cloudlet Based Mobile Computing," in *Proceedings of the 2015 International Conference on Cloud Computing and Big Data (CCBD '15)*, pp. 268–275, IEEE, 2015.

[30] Y. Jararweh, A. Doulat, O. AlQudah, E. Ahmed, M. Al-Ayyoub, and E. Benkhelifa, "The future of mobile cloud computing: Integrating cloudlets and Mobile Edge Computing," in *Proceedings of the 23rd International Conference on Telecommunications (ICT '16)*, pp. 1–5, IEEE, 2016.

[31] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "OpenStack: toward an open-source solution for cloud computing," *International Journal of Computer Applications*, vol. 55, no. 3, pp. 38–42, 2012.

[32] A. Solano, R. Dormido, N. Duro, and J. M. Sánchez, "A self-provisioning mechanism in openstack for IoT devices," *Sensors*, vol. 16, no. 8, 2016.

[33] V. Herwig, R. Fischer, and P. Braun, "Assessment of REST and WebSocket in regards to their energy consumption for mobile applications," in *Proceedings of the 2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS '15)*, pp. 342–347, IEEE, 2015.

[34] G. Merlino, D. Bruneo, F. Longo, S. Distefano, and A. Puliafito, "Cloud-Based Network Virtualization: An IoT Use Case," in *Proceedings of the International Conference on Ad Hoc Networks*, pp. 199–210, Springer, 2015.