Edge Computing Framework for Distributed Smart Applications

Kaikai Liu, Abhishek Gurudutt, Tejeshwar Kamaal, Chinmayi Divakara, Praveen Prabhakaran
Computer Engineering Department
San Jose State University (SJSU)
San Jose, CA, USA
Email: {kaikai.liu, abhishek.gurudutt, tejeshwarchandra.kamaal, chinmayi.divakara, praveen.prabhakaran}@sjsu.edu

***Abstract*—The rapid growth in technology and wide use of internet has increased smart applications such as intelligent transportation control system, and Internet of Things, which heavily rely on an efficient and reliable connectivity network. To overcome high bandwidth work load on the network, as well as minimize latency for real-time applications, the computation can be moved from the central cloud to a distributed edge cloud. The edge computing benefits various smart applications that uses distributed network for data analytics and services. Different from the existing cloud management solutions, edge computing needs to move cloud management services towards distributed heterogeneous edge nodes for multi-tenant user applications. However, existing cloud management services do not offer remote deployment of multi-tenant user applications on the cloud of edge nodes.**

In this paper, we propose a practical edge cloud software framework for deploying multi-tenant distributed smart applications. Having multiple distributed end nodes, auto discovery of all active end nodes is required for deploying multi-tenant user applications. However, existing cloud solutions require either private network or fixed IP address, which is not achievable for the distributed edge nodes. Most of the edge nodes connected through the public internet without fixed IP, and some of them even connect through IEEE 802.15 based sensor networks. We propose to build a software platform to manage the distributed edge nodes as well as support services to deploy and launch isolated, multi-tenant user applications through a lightweight container. We propose an architectural solution to remotely access edge cloud management services through intermittent internet connections. We open sourced our whole set of software solutions, and analyzed the major performance metrics of the edge cloud platform.

Edge Computing, Fog Computing, Internet-of-Things, Multi-tenant Applications

I. Introduction

As predicted by Cisco Internet Business Solutions Group, by 2020, 50 billion devices will be connected to the Internet [1]. Data produced by these devices will reach 500 zettabytes, according to Cisco Global Cloud Index, by 2019. It is difficult to provide such high bandwidth since the capacity of global data centers will only reach 10.4 zettabytes by the year 2019 [2].

Edge computing is a technology that allows the computation taking place at the vicinity of data being produced on distributed micro "data centers". If the IoT-generated data is processed, stored, analyzed, and operated close to, or at the edge of network, the problem of high network bandwidth requirements for future IoT applications will be solved. Moreover, the edge computing node acts as both consumer and producer, which reduces the data processing pipeline and improves the event response time. For future IoT applications, e.g., Smart City monitoring, traffic sensing, security enforcement, fast response is really the key to ensure smart city service qualities. However, to enable edge computing for smart city applications, there are multiple challenges need to be solved, for example, remote device programmability, naming, data abstraction, service management, privacy, security, and other optimization metrics. Programmability addresses the problem of deploying an application compatible to heterogeneous end node platforms. Naming conventions in Edge computing has to be implemented for things identification, programming, data communication, and addressing since multiple end nodes and applications would be present. Data abstraction is required since huge data would be generated, but only required data should be filtered and also abstracting useful data from unreliable sources is a challenge. Service Management, is to provide a framework to manage all the edge nodes from a remote location. Privacy and Security, is a major concern for smart home or smart health applications, which generates high amount of sensitive data.

Several of the existing edge computing platform supports either a single-user application cloudlet, application specific cloudlet or a single service oriented cloudlet. The extension of Openstack for cloudlet (Kiryong Ha, Mahadev Satyanarayanan), Openstack++ provides multi-tenancy through VM based computing resources. This solution of providing hypervisor based computing resource to establish multi-tenancy adds redundant software abstraction layer to give a non-deterministic application. The sensors required by these applications talk to the edge device node over wireless WAN, BLE, ZigBee etc. There can be many sensors that is required by multitude applications connected to a single edge node and each of these application can be maintained by third-party vendors experimenting in that domain. For example, a single edge node can be connected to a surveillance video sensor to compute data for an intelligent surveillance application, and the same device can also gather air quality data from connected sensors to provide real-time data for a smart life system. Abstracting the hardware edge node to multi-tenant applications can help improve the device utilization, lower

the system deployment cost, and accelerate the real world test and deployment of new applications. However, to support multi-tenant 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 multi-tenancy; 3) creating a versatile runtime environment that supports variety of technologies and devices from different vendors; 4) providing a remote dashboard of the edge cloud that enables third party application vendors/users to launch and manage applications remotely.

In this project, we propose a software framework to enable the envisioned edge cloud platform with three key functionalities: 1) supporting multi-tenant isolated user applications via lightweight virtualization; 2) providing cloud dashboard for remote programming and application management; 3) featuring robust network connectivity that agnostic to network mediums. The framework provides computing resources through a lighter, more deterministic container based virtualization (IaaS). The computing resource holds all the dependent libraries, runtime environment required to run the isolated application (PaaS). The edge cloud provides a solution offering remote dynamic discovery that enables user to control and manage applications remotely through a web application dashboard (SaaS). Existing cloud solutions require either private network or fixed IP address, which is not achievable for the distributed edge nodes. Most of the edge nodes connected through the public internet without fixed IP, and some of them even connect through IEEE 802.15 based sensor networks. We propose to utilize resilient messaging tunnel to manage the distributed edge nodes that make it agnostic to network mediums as well as supporting dynamic access with mobility support.

II. System Overview

A. System Design

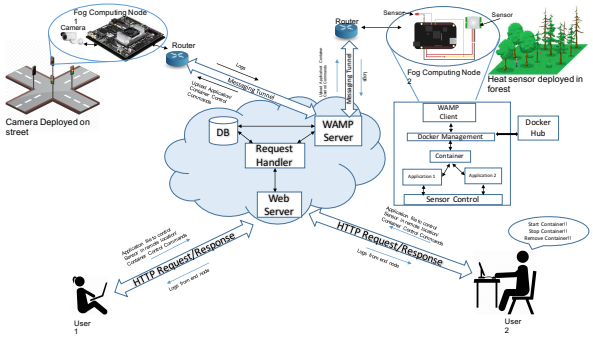


Fig. 1. Edge Computing Architecture Diagram.

As shown in Fig. 1. The proposed edge cloud framework consists two major parts: the cloud side and the edge node side. The cloud side includes user dashboard for remote management, message routing, and database. The edge node part constitutes of messaging, device monitoring, virtualization and container management.

1) Cloud Dashboard: As shown in Fig. 2, 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, then stored to the database to maintain a copy.

User Interface. The user interface provides 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, the user 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 has become an important portion of Internet of Things. Messaging services not only include communication between two nodes, but also has to 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 server, and the secure communication, is fulfilled with the help of our proposed messaging service. We utilized the Web Application Messaging Protocol (WAMP) to implement the message service and make it to support the node mobility with dynamic IP.

Database. Database will be used to save information related to user, container, location of the stored results and so on. We utilize relational database to establish relationship between different management entities. To achieve modularity, the database is divided into 5 components, i.e., user, compute, image, device and storage. A relationship is established between the modular DB tables to link the user to the resource 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 and offers a loose coupling with other service modules.

2) Edge Node: **Container Management**. We utilize the container as the lightweight virtualization technique for the edge node. Containers is a software that is built with complete file system which consists of system libraries, application program and so on. It is lightweight when compared to virtual machines and these run using same operating system kernel and use less RAM. These features will enable the low-power and low-cost edge node to support dense applications with less cost.

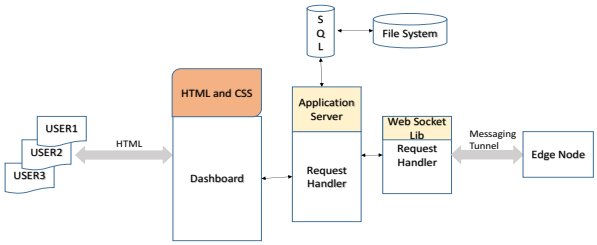


Fig. 2. The server architecture of the edge platform.

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 manager the containers deployed on a machine such as Swarm, Kubernetes, Fleet and so on. Swarm container management is technique is very effective when compared to other techniques. It converts all the Docker hosts into one single virtual host. All the nodes created on the edge device will be controlled, scheduled, monitored and cleared using swarm manager. Swarm manager will also help 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 will be installed before executing an application. Every user can deploy new container for each application and later obtain the results. Once the execution of an application is completed, then containers are deleted automatically by the Docker daemon.

Messaging service. WAMP (Web Application Messaging Protocol) is used in the end node to receive messages from the server. Commands received from the server are forwarded to container management. Messages such as heartbeat, status and registration are transmitted to server.

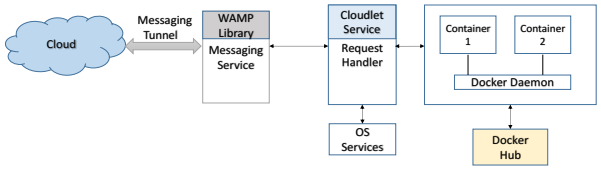


Fig. 3. The architecture of the edge node platform.

III. Cloud Orchestration Framework for the Remote Edge Node

A. Resilient Network with High Mobility

Existing cloud solutions require either private network or fixed IP address, which is not achievable for the distributed edge nodes. Most of the edge nodes connected through the public internet without fixed IP, and some of them even connect through IEEE 802.15 based sensor networks. We propose to build a software platform to manage the distributed edge nodes through a lightweight messaging services. 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 the power when necessary.

We utilize the WAMP, which is an open standard Websocket protocol, to provides both remote procedure calls and publish-subscribe messaging services. WAMP is mainly used for distributed, multi-client (edge nodes in our use case scenario) and server application. Autobahn framework and Crossbar are used to host a server and for messaging routing. Autobahn is an open-source implementation of WAMP and Websocket. This provides libraries which can be used to create client and server application. In this project, python version of library is used. Crossbar is a WAMP router is configured on the server to direct the messages to subscribed clients. Crossbar, a networking platform for distributed and micro-service application is used in our application to handle routing.

In a publish & subscribe (PubSub) approach, the server and client are coupled through a router to Publish an 'abstract' topic to the Subscribed client. After transmission is complete, the server and client are decoupled, by the router, also named broker. Broker keeps a track of subscriptions. When an 'event' is published by a Publisher to a topic, the Broker looks up the record maintained to determine on the list of Subscribers subscribed on that topic, and then forward the information ("event") to all those Subscribers. This act of dispatching the information to determined receivers is called routing.

B. Cloud Dashboard

In the cloud dashboard, user registration is the first step, upon which the user can login to the user dashboard to orchestrate the end nodes. Various commands such as start, stop, create, and remove a container, upload and download a file from a virtual space on the end node is provided for the user.

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

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

Device ID generation. When the end node application is started, device name and MAC address of the device is fetched. The same is concatenated in the form 'device name / MAC address'. With the help of both device name and MAC

address, it is possible to keep each device ID unique, helping to differentiate the message to be published according to the device selected by user.

Location of device. The location of the device is fetched based on the IP address to which it is connected. This helps in classifying each device based on the location. Smart city based applications can make use of this feature to segregate each installed device.

Architecture of the device. The architecture of the end node is fetched and is updated during the device registration. Containers compatibility is based on the architecture, hence by detecting the architecture, suitable containers can be listed to the user for deployment. On successful login user can access his/her dashboard. In the dashboard user can find his list of containers created all the devices. The user can also control the containers by clicking the container control buttons. The user can also see the device information by selecting the device.

C. Server Side messaging service

. **Handling Registration and Heartbeat**. Every end node which connects to the server is registered with a unique ID; to help identify PubSub to/from a particular device. The unique ID would be part of the topic used while subscribing. When a user requests to perform actions on the end device, this unique ID is used in the URI, to Publish the request to desired end node. Periodic heartbeat message is transmitted to the server from all the registered end node, to make sure the end device is available for the user. The heartbeat message will contain information about 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 database upon registration. Upon heartbeat stop, the corresponding device is removed from the database along with the containers on the device and the CPU information.

Handle User Request. The requests from the user is received through a REST API. The requests which is intended to 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, a 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 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.

IV. Edge Node Virtualization

A. Edge Node

End node, upon boot up registers to the server with a device name and transmits periodic heartbeat to indicate the end node is alive. End node also transmits CPU information and container status periodically to server. End node and user information is maintained in a database. Client side contains an edge node. An edge node can be any embedded device such as Beaglebone Black, Raspberry Pi, Jetson TX1 and so on.

Docker engine is installed on the edge node which allows the users to deploy their application by creating containers. Docker provides numerous amounts of packages for developers to design and develop application programming interface for creating, deleting, running an application and so on. 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, Docker receives a command from the WAMP agent and performs the necessary action.

Heartbeat. Once the device boots up, the device checks for an internet connection, and reboots to try 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 server. The periodic heartbeat indicates the end node is alive.

CPU info. The CPU info fetches the device name, memory consumption, CPU usage, disk memory available, operating system on end node, kernel version and is updated to the server periodically.

Container Status. The status of the container is updated periodically. The status message contains the container name, status, device name to detect the device maintaining the container, and image name. All the status message is updated to database and is notified to user based on the username.

Registering and Heartbeat. Once the device boots up, a registration message is transmitted to the server. Registration message consists of device ID. The device ID would be unique to each device. The device will subscribe to all the 'events' with a unique URI containing the device ID. After registering, device would send a periodic heartbeat. Heartbeat would stop when the device turns off, hence informing to unregister from the server and also informs user about device being no more available for usage. Device ID would be transmitted along with the heartbeat.

CPU info and Container Status update. The CPU info and container status are fetched periodically and the same is updated to server. CPU information can be used to detect the available system resources before deploying containers. Container status can be redirected to inform the user about the status of each container.

Handle Request. The subscribed 'event' receives the request from the server, containing information about container control. The received request is then parsed to the format as required by the Docker API. During 'create' request, each container is renamed to the user's demand. Container name received through the request, is further formatted to fit in the form 'username-containerName', by doing so, each container created will have 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.

File Download. The application file is downloaded from the user dashboard to the end node when user requests. The user request, is validated on the server and then passed on to respective end node. The received commands contain the location of application file on the server which is used for download.

V. Evaluation

The performance evaluation is performed by deploying a server on AWS and considering Beaglebone Black as the end node. The following performance is impacted by the 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.

A. Launch Containers in the Edge Node

Performance for creating a remote container by downloading a new image from the cloud. Fig. 4 shows the performance for creating a remote container by downloading a new image from the cloud. This is the first step when the user want to deploy their application to the remote edge node. The user can select the type of the container image and push it the edge node. The Fig. 4 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 considered different 5 images with size ranging from 1.5kB to 350MB. The calculated time includes the downloading time of the image and the time of creating a container out of it. Network latency is assumed to be equal during the course of evaluation.

To pull hello-world image, which is of 1.8kB in size from Docker hub and create a container it took approximately 3.75 secs. Upon issuing the command, Docker daemon initials looks for the image in the cache and if doesn't find it then 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 additional to the linear downloading time. Fig. 4 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 image. Fig. 5 represents the time required by the Docker daemon to create a container using an image stored in cache. When user requests to create container with a specific image, the application passes the image name and the container name to the Docker daemon. Docker daemon 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

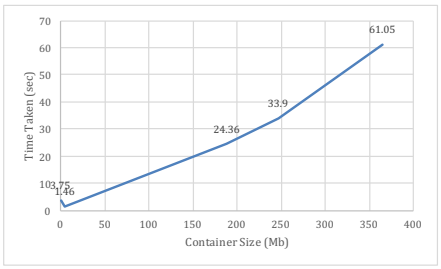


Fig. 4. The time taken to create container for new images with different size.

measured here is the comparison between the time taken to create a docker container with the help of docker and the time taken to processing all the commands received from the user. To evaluate the performance, we downloaded 5 images with

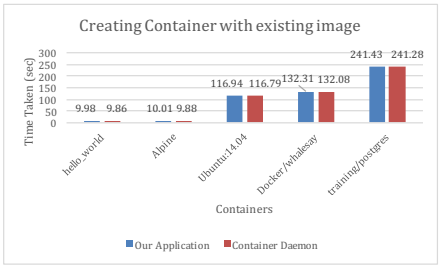


Fig. 5. The time taken to create container for existing images with different size.

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

System overhead and 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 to different hardware platforms. This flexibility is very important in utilizing different hardware platforms from different vendors to create a common edge computing platform for multi-tenent 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, 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 multi-tenancy support. Specif-

ically, 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; 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 8 containers, the system was overloaded and slowed down. Hence it is recommended to use light weight containers with 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.

B. Upload Applications to the Edge Node

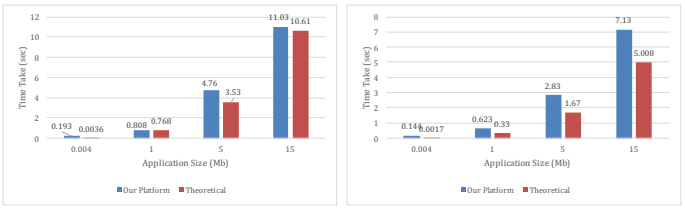
Our dashboard provides functions for users to upload their applications to the remote containers in the edge node. The applications can be the source code, complied executable file, or jave compressed packages. To facilitate multiple application file requirement, we utilize a common format (.tar) for 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 user to the edge node through our messaging service layer by two steps: 1) transfer the application file to the web server; 2) transfer the application file from the web server to the edge node. The file size varied from 4 KB to 15 MB. The performance analyzed here is in comparison to theoretical values calculated by the formula "file size $\times$ 8 / network bandwidth". Overhead of 50% is added for file size lesser than 1 MB and 10% for file size greater than 1MB.

Transfer the application file to the web server. To evaluate the performance of uploading an application file from user dashboard to webserver, we perform time consumption calculation 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 a HTTP request. The uploaded file is then moved to a unique storage space with unique ID. When the file is too big, we will divide them into several small compressed files, then these compressed files will be transferred. The total time consumed includes the file compression, division, and transfer. As shown in Fig. 6a, the results show that the time consumed was almost linear with 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.

Transfer the application file to the edge node. Application file uploaded by user are to be transferred from user dashboard to the remote node and install 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 chunk of data stored on the server and transmitting to end node serially through a HTTP request. Received chunk of data is stored on to 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 files were recorded. The file size varies from 4 KB to 15 MB. As shown in , 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.



(a) To the web server (b) To the edge node

Fig. 6. The time taken to transfer the application file: (a) to the web server; (b) to the edge node.

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, 2.6GHz clock, 8GB physical memory. As shown in Fig. 7, it was observed that there was negligible amount of time difference when performed in different platforms.

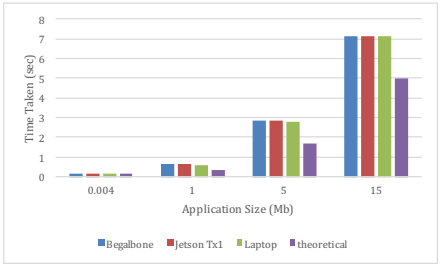


Fig. 7. The time taken to transfer the application file to the edge node in different hardware platforms.

C. Sending Command and Debugging the Edge Node

Sending command and getting response. The WAMP messaging protocol was made use in our platform to communicate between server and end node, for example, transmitting a command from user dashboard to end node and get the

response. WAMP protocol is an asynchronous communication protocol, which is an added advantage to our platform. Our platform made use of publish and subscribe feature of WAMP. While transmitting a command from server to end node or while transmitting a response from end node to server, commands and response are noted to be in a string format. The string size differs for various commands and response. It was observed that the time taken to transfer a string of 56 characters (56 bytes), was around 1.68 milliseconds. Different commands from the user was translated to a string which constituted of different size, 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 end node to server was timed and it was observed that the time consumed was in the range of 2.53 milliseconds to 2.84 milliseconds.

Getting the log file from the edge node. As user requests

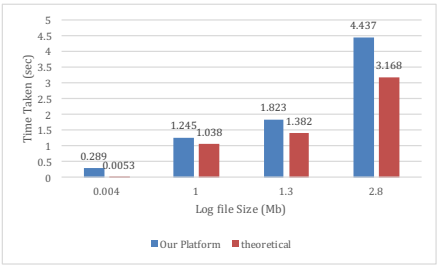


Fig. 8. The time taken to transfer the log file from the edge node to the cloud dashboard.

to fetch the log file for the debugging purpose, a command from the cloud dashboard will be transferred to the end node, and fetches the log result file from end node to the cloud dashboard. The transfer of file from end 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 end node to cloud dashboard. The received chunk is copied onto a file on the server until the end of file. Different sized result files were considered to measure time and the time taken. Upload speed of 11.45Mbps is considered during the evaluation. Fig. 8 shows the required time to fetch the log file with respect to different file sizes. The difference in time is due to the time taken by our platform to send a message from webserver to end node, process the request, copy the file from the container to the local path, locate the file and then transfer to the cloud. This analysis measures the time required for the complete procedure which constitutes of, send a command from user dashboard, route the command to a particular edge node, copy the file from container to the local path on the edge node, transfer the file from to the server, point to the messaging service lay to download and transfer of file to the user dashboard.

D. Comparison of Our Platform with Other Solutions

As shown in Table I, when comparing with existing solutions, for example, Stack4Things [3], [4], the advantage of our solution includes multi-tenancy support in the remote edge node, compatible to various applications, and allow remote access as well as remote debugging and data transfer.

TABLE I
COMPARISON OF OUR PLATFORM WITH STACK4THINGS

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

The second type of the related solution is VPN-based cloud solution. They utilize VPN-based solution 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. VPN based remote access mechanism is one of commonly used approach to access a remote node on a private network. For example, existing cloud solution requires private network with fixed IP address to connect peer computing node. Leveraging the VPN connection, the private network can be expended to the network edge. It works by creating a secured network connection over public network like the Internet. VPN can connect multiple sites over large distance typical to 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 other VPN server, distributed in various geographical sites where the edge device is deployed. All the device deployed will be part of a virtualized private network. However, this kind of solution has multiple limitations: 1) does not support dynamic edge node mobility; 2) high overhead in terms connecting remote edge nodes; 3) not robust to link failure.

To compare our approach with VPN-based solutions, we deploy 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 and client to measure the upload and download throughput. As shown in Fig. 9, 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.

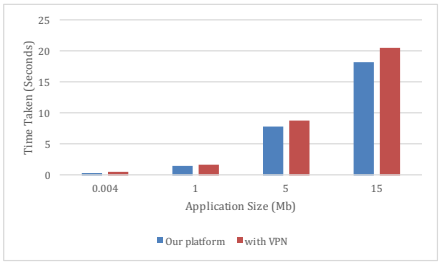


Fig. 9. The performance comparision with VPN based approaches.

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

TABLE II
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 is uses WAMP messaging	VPN connectivity relies on high bandwidth as it channels the network over the Internet

VI. RELATED WORK

The existing cloudlets can be classified based on the services that is offered by them. These services are based on application associated with the edge like IoT or it can even be special services that is needed by the network like security policy, content delivery etc. Some of the major players in the edge cloud are companies like Cisco and Akamai [5], [6], [7], [8]. Akamai cloudlets provide vendor application that are designed to solve specific business and operational challenges [9], [3].

Openstack [10], [11] 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. Though this platform a user can own multiple virtual computing instances, each configured for an application environment that runs independent isolated applications. Openstack consists of interrelated components each component offering its services through RESTFul API's. It provides a web user dashboard that connect the services offered by components involved in computing, storage and network management. Openstack can be extended to cloudlet at network edge to provide robust services [12].

VII. CONCLUSION

This paper proposes an edge computing oriented platform solution for developers to remotely orchestrate IoT devices without caring about their physical location, or their network configuration. Leveraging the minimal usage of network bandwidth by asynchronous communication between server and client, we enable developers to deploy applications in a virtualized space, debug the remote application, 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. Containers consume less memory when compared to a virtual machine, which is an advantage due to the memory constraints in embedded environment. These containers can communicate over the network which allows us to remotely orchestrate it through a dashboard. Our platform is useful in applications that uses remote sensing and monitoring, having bandwidth limitations and also in some applications where there is no need for continuous connection to be established between user and end node. We open sourced our solution through Github namded "Edge-computing-embedded-platform". In the future research direction, we will further reduce the overhead when deploying the applications. Specifically, we will enable secure shell option to a specific container, which allows the developer to have the fine grained control of the container. Our platform current doesn't provide this feature due to restricted permissions to access port on end node. Also, to implement this feature, port mapping mechanism in server has to be added. We will also attach multiple sensors to the end node, with each sensor can be shared by multiple containers. A feature on the dashboard to display the sensor mapped to the specific hardware and container will be provided.

ACKNOWLEDGMENT

The work presented in this paper is supported in part by National Science Foundation under Grant No. CNS 1637371.

REFERENCES

[1] D. Evans, "The internet of things how the next evolution of the internet is changing everything (april 2011)," *White Paper by Cisco Internet Business Solutions Group (IBSG)*, 2012.

[2] C. V. Networking, "Cisco global cloud index: Forecast and methodology 2014-2019 (white paper)," 2013.

[3] G. Merlino, D. Bruneo, S. Distefano, F. Longo, and A. Puliafito, "Stack4things: integrating iot with openstack in a smart city context," in *Smart Computing Workshops (SMARTCOMP Workshops), 2014 International Conference on*. IEEE, 2014, pp. 21–28.

[4] F. Longo, D. Bruneo, S. Distefano, G. Merlino, and A. Puliafito, "Stack4things: An openstack-based framework for iot," in *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*. IEEE, 2015, pp. 204–211.

[5] A. V. Dastjerdi, H. Gupta, R. N. Calheiros, S. K. Ghosh, and R. Buyya, "Fog computing: Principles, architectures, and applications," *arXiv preprint arXiv:1601.02752*, 2016.

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

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

[8] Z. Pang, L. Sun, Z. Wang, E. Tian, and S. Yang, "A survey of cloudlet based mobile computing," in *Cloud Computing and Big Data (CCBD), 2015 International Conference on*. IEEE, 2015, pp. 268–275.

[9] 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 *Telecommunications (ICT), 2016 23rd International Conference on*. IEEE, 2016, pp. 1–5.

[10] 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, 2012.

[11] A. Solano, R. Dormido, N. Duro, and J. M. Sánchez, "A self-provisioning mechanism in openstack for iot devices," *Sensors*, vol. 16, no. 8, p. 1306, 2016.

[12] K. Ha and M. Satyanarayanan, "Openstack++ for cloudlet deployment," *School of Computer Science Carnegie Mellon University Pittsburgh*, 2015.