The Journal of Systems and Software 119 (2016) 149–161

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Design and implementation of a novel service management framework for IoT devices in cloud

Chinmaya Kumar Dehury, Prasan Kumar Sahoo*

Dept. of Computer Science and Information Engineering, Chang Gung University, Kwei-Shan, 33302, Taiwan(ROC)

ARTICLE INFO

Article history:
Received 8 March 2016
Revised 21 May 2016
Accepted 22 June 2016
Available online 24 June 2016

Keywords:
Cloud computing
SaaS
IoT
Docker

ABSTRACT

With advent of new technologies, we are surrounded by several tiny but powerful mobile devices through which we can communicate with the outside world to store and retrieve data from the Cloud. These devices are considered as smart objects as they can sense the medium, collect data, interact with nearby smart objects, and transmit data to the cloud for processing and storage through internet. Internet of Things (IoT) create an environment for smart home, health care and smart business decisions by transmitting data through internet. Cloud computing, on the other hand leverages the capability of IoT by providing computation and storage power to each smart object. Researches and developers combine the cloud computing environment with that of IoT to reduce the transmission and processing cost in the cloud and to provide better services for processing and storing the realtime data generated from those IoT devices. In this paper, a novel framework is designed for the Cloud to manage the realtime IoT data and scientific non-IoT data. In order to demonstrate the services in Cloud, real experimental result of implementing the Docker container for virtualization is introduced to provide Software as a Service (SaaS) in a hybrid cloud environment.

© 2016 Elsevier Inc. All rights reserved.

1. Introduction

Internet of Things (IoT) is the upcoming technology to convert an object into smart objects by connecting them to internet. This allows us to control any tangible object remotely. The term IoT coined (Epc symposium, 2003) for supply chain management and came to the attention when Auto-ID center launched their initial vision of the epic network for automatically identifying and tracing the flow of goods in supply-chains. According to an estimation made by Cisco, 50 billion devices will be connected to internet by 2020 forming a large dense IoT environment (Internet of things , iot). These IoT devices include intelligent oven, smart watch, smart air quality monitor, heart monitoring implants, rescue operation etc. Different sensors such as temperature sensor, gyro sensor, accelerometer, pressure sensor, and humidity sensors play an important role in establishing IoT environment. From above discussion, the basic definition of IoT can be derived as worldwide collaboration and formation of massive network among all physical entities or objects to reach common goals through unique addressing schemes by embedding RFID tags, sensors, actuators etc. As defined by the International Telecommunication Union (ITU) (Itu-t recommendation database, 2016), IoT is a global infrastructure for the information society, enabling advanced services by interconnecting physical and virtual things based on existing and evolving inter-operable information and communication technologies.

In Atzori et al. (2010), authors discuss three main visions of Internet of Things. The Semantic-oriented vision states that the increasing number of IoT devices increases the complexity in handling those devices. Huge amount of data are being generated in every instance of seconds, and the data exchange among IoT devices also increases exponentially. The semantic-oriented vision addresses the aforementioned issue, where the representations of huge data, storage, search, and organize information become cumbersome. The idea behind semantic-oriented vision is to propose new semantic technologies, which can exploit new solutions to handle such massive amount of data generated by IoT devices. The Internet-oriented vision states that distributing unique IP to each object to connect to the internet is difficult and inefficient as the number of IoT object increases. The existing internet protocol may not be able to handle billions of smart IoT objects. Hence, how to connect several objects to the internet and how to identify uniquely the data generated from those objects is the main concern in internet-oriented vision. Things-oriented vision mainly refers to the tracking of each object with specialized technology such as RFID tag. RFID tag is most popular technology in

* Corresponding author.
E-mail addresses: d0321009@stmail.cgu.edu.tw (C.K. Dehury), pksahoo@mail.cgu.edu.tw (P.K. Sahoo).

http://dx.doi.org/10.1016/j.jss.2016.06.059
0164-1212/© 2016 Elsevier Inc. All rights reserved.

establishing an IoT environment. But things oriented vision indicates researchers to go beyond the RFID tag.

Cloud computing is a computing paradigm, where all computing resources such as memory, storage, processing capabilities are rented from third party provider through internet. This paradigm is unlike the traditional way of computation, where the users have complete set of resources in a standalone personal computer to perform any computation task. In cloud computing, all the resources are accessed through web application and it is a potential platform for different companies, organizations, individuals, researchers for its capability to scale up and scale down the resources as per the requirement. Mainly, the computation power and the storage power are the two features cloud computing platform focuses on. Everything cloud provides to the users is called as service. These services are of three types. Software as a Service (SaaS) at the top of the control hierarchy gives users minimal control over the cloud resources. The users can use the rented software only and the underline environment is kept hidden from the users. On the other hand, Platform as a Service (PaaS) facilitates users to customize the platform, which includes the application and middleware, to develop own application. Infrastructure as a Service (IaaS) gives users full control to customize the environment for developer. The customization includes the application, development platform, middleware, the operating system and the required resources such as mefmory, storage, CPU etc. Here, the users have no idea regarding the underline environment such as the hypervisor, physical server etc. All the services are deployed in any one of the four different recommended models; Private cloud, Public cloud, community cloud and hybrid cloud. Different models are categorized solely based on the concern targeted user.

The underlined technology that makes the cloud environment possible is virtualization (Gurav and Shaikh, 2010). Virtualization technology allows the physical resources to be shared by multiple users without the knowledge of each other. The cloud computing environment is becoming matured as virtualization technology becoming more matured. This technology enhances the efficiency in resource utilization and power consumption (Qian et al., 2015). In cloud environment, the physical resources are provided to the users in the form of virtual machines. These virtual machines act like physical machines to the users and users have the capability to customize the virtual machine at any instance of time. On the other hand, the alternative to hypervisor based virtualization technology, Linux container technology is evolving as potential technology to provide SaaS (Li et al., 2015). Container based virtualization is lightweight virtualization technology, enabling high resource utilization and less overhead, suitable for hosting software (Joy, 2015).

The rest of the paper is organized as follows. Brief summary of related literature and motivation behind our proposed work are given in Section 2. The proposed framework for managing the realtime data in Cloud is designed in Section 3. Performance analysis of our proposed framework is given in Section 4. Implementation of Docker platform for virtualization in Cloud is described in Section 5. Performance evaluation of our framework and experimental results of Docker platform are given in Section 6 and concluding remarks are made in Section 7.

2. Related works

Applying cloud computing concept to IoT environment is becoming mostly discussed research area now-a-days. These two technologies can be applied to diversified environment such as home security, emergency control, traffic control, building management, smart cities, health care system, fire detection, dam monitoring system etc. Authors around the world are proposing their views in merging power of IoT and cloud computing in different environments. For example, in Yu et al. (2015) authors propose the application of IoT in cloud based building management. The goal of this application is to reduce the energy consumption by building management system with realtime building energy forecasting. Similarly, in Suciu et al. (2015), authors propose e-health application of IoT and cloud computing in convergence with big data. The volume of data generated by IoT devices in health care system is considered as big data and hence it gives many challenges such as storage and processing. According to authors, Machine to Machine (M2M) communication enabled by IoT paradigm not only plays an important role in tele-monitoring e-health architecture, but also makes the architecture secure. However, the major problem of this architecture is that the realtime emergency tasks are not processed with highest priority. The model architecture for remote monitoring cloud platform of healthcare information (RMCPHI) (Luo and Ren, 2016) is designed for managing the healthcare information by prioritizing and scheduling the realtime emergency tasks. However, the proposed framework does not give any insight how to prioritize the tasks. Besides, execution of emergency tasks before deadline is not emphasized, which is a major pitfall of this framework.

Authors in Schaffers et al. (2011); Vlacheas et al. (2013); Zanella et al. (2014) propose their architecture for building the socio-economic developed and secure smart city by applying IoT and cloud computing paradigm. Authors also emphasize on different issues such as heterogeneity among IoT devices, unreliable nature of associated services etc. The current discussion on application of IoT and cloud computing in building smart city is heading towards developing urban IoT architecture, which can handle a variety of heterogeneous connected IoT devices and different types of generated data form those connected devices. IoT can be combined with cloud computing considering a generalized environment as discussed in Li et al. (2013); Bai and Rabara (2015); Khodadadi et al. (2015) with different research issues. Authors in Li et al. (2013), propose the IoT service delivery on cloud in efficient manner that can be scaled up and scaled down as per the requirement. Domain mediator is proposed to handle the heterogeneity of IoT devices. However, although the utilization of IoT devices are optimized by sharing them among multiple users, a little attention has been paid how to handle the users who need less computation resources for few time period. A data-centric framework for IoT applications in cloud (DCFIC) (Khodadadi et al., 2015) is proposed, which can handle data filtering by balancing the load and scheduling the cloud resources. However, it is not clear how DCFIC can handle the workload of real and non-realtime data requests. Further, the framework does not emphasize on the computation intensive requests.

Meanwhile the power of cloud computing is applied to supply chain management in order to improve the inter-operability and efficiency among stake holders as discussed in Yinglei and Lei (2011); Liu et al. (2013). In Yinglei and Lei (2011), authors propose cloud computing based architecture for supply chain management. The traditional legacy applications are provided through cloud provider online. The proposed architecture takes advantages of SaaS model to provide services to its user. In Liu et al. (2013), authors propose trusted model of cloud based supply chain management system where they consider six trust factors such as communication, learning capacity, participation, usability, security and reliability. Pharmaceutical industrial context was considered to evaluate the proposed trust model. On the other hand, authors in Leukel et al. (2011) propose another service model inspired from supply chain system. They map the basic cloud computing concept to the basic concept of supply chain system. The goal of supply chain system is to effectively handle all vehicles while efficiently using the available resources. From this basic goal, authors proposed Supply Chain as a Service (SCaaS) to reach the goal of efficient utilization of resources and delivery of Quality of Service without compromising Service Level Agreement.

Authors in Jiang et al. (2014) design a storage framework for IoT devices in cloud, which combines multiple databases with Hadoop and enables the cloud service provider to handle the diversified data and store them in a distributed manner. Considering processing capability of the cloud computing, authors in Wang and Ranjan (2015) introduce a brief summary on currently available solutions in processing huge amount of distributed data followed by research issues associated with processing of those distributed IoT data. Security is another major factor need to be taken into account while integrating IoT with cloud. Authors in Suarez et al. (2016) propose a secure IoT data management architecture considering other requirements such as naming, inter-operability and energy-efficiency. However, though data storage or data processing is considered in most of these works, priority of the users to maintain fairness and Service Level Agreement (SLA) among users are not taken into account, which are the major drawbacks in the above mentioned works.

The study in Lee et al. (2014) illustrates the architecture and design principles of vehicular cloud networking that handles the data generated by different sensors fitted within a vehicle. Considering road safety and passenger comfort, another generic cloud computing model (Bitam et al., 2015) is proposed for Vehicular Ad hoc Networks (VANET). In Mallissery et al. (2015), authors design a secure VANET cloud application to ensure minimum traffic congestion, traveling time, accidents and environmental pollution. However, the authors have not emphasized on the real and non-realtime data requests generated from different vehicles. Besides, it is found that proposed frameworks do not consider the requests on priority basis. Apart from aforementioned applications such as smart cities, health care system, and building management, the capability of IoT and cloud computing has also been applied to dam monitoring and pre-alarm system (Sun et al., 2012), in-home environment condition monitoring (Kelly et al., 2013), vehicular data cloud service (He et al., 2014) etc. However, in the above mentioned related works, authors have not considered the realtime data analysis and service models for the cloud. Based on the above literature survey, we present the motivation behind our work as follows.

2.1. Motivation

In this subsection, we present the motivations behind our work related to how to design the cloud platform that can handle the realtime data generated by the IoT devices and batch data for the scientific computation and storage. Although, IoT is the answer for many issues while making human life more comfortable and luxurious, this also introduces many challenges. Storage and computing capability are the major challenges of IoT devices such as sensors. At the same time, Cloud computing addresses these two issues for the data generated by IoT devices. IoT devices are mainly responsible for generating huge amount of data. As we discussed earlier, in real world IoT devices are of many kinds. For example, the sensors that are fitted in human body, monitor different factors of patients such as monitoring hemoglobin in blood, body temperature, etc. Such kinds of IoT devices, generate very small amount of data but very frequently. On the other hand in surveillance system, the cameras that are setup to detect the intrusion generate large volume of data very frequently. Some IoT devices generates huge amount of data, but in specific time interval. Our basic concern is that the nature of data varies from one device to another based on the environment. The application specific data are diversified by their nature, type, volume and variety. We assume here that those diversified data needs special attention unlike the conventional way of handling the huge amount of data.

In future, we will experience different types of IoT applications in our daily life. Different specific applications enable our life to

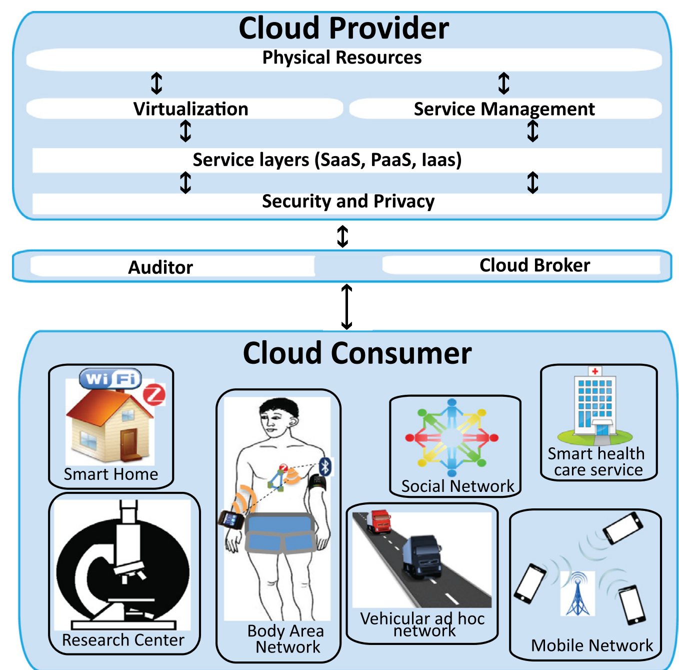


Fig. 1. The proposed Cloud Framework.

be more comfortable, automatic and luxurious. But, from cloud service provider (CSP) point of view, this may introduce another level of complexity as they may need to handle different applications or different environments at the same time. For example, CSP may need to provide the resources for many smart services such as smart cities, emergency control, smart health care system, smart traffic control system, smart home security etc. In many applications, IoT devices generate data on realtime basis. Those data need to be stored time to time and they also need to be processed on realtime. This motivates us to think towards combining the power of IoT technology and Cloud computing paradigm to build an underlined framework that cannot only handle different smart services or smart environments but can also utilize the available resources efficiently to maximize the revenue for cloud service provider.

3. Proposed cloud framework

In this section we design a cloud framework SMFIC (Service Management Framework for IoT devices in Cloud) to handle the realtime data generated from various IoT devices and social media as well as the non-realtime data for scientific computation and storage. The basic architecture of our SMFIC is depicted in Fig. 1. As shown in the figure, the proposed SMFIC framework can be divided into three different layers with five major components. From the application and service point of view, the three layers are the Consumer layer, Service Provider layer and the Middle layer that brokers the services and available resources between the consumer and service provider and works on demand and supply basis. The five different components reside in those three layers and are known as the actors. The actors in this architecture are cloud service provider, cloud broker, cloud auditor, cloud consumer, and cloud carrier. In the background, Cloud carrier plays a vital role as backbone of the cloud environment as all communications are made through the cloud carrier. Cloud provider also known as the cloud service provider and cloud consumer are the two end of a line. Cloud service provider (CSP) provides the physical resources on rented basis in the form of services such as IaaS, PaaS, and SaaS. All the services are meant for cloud consumer. Every time cloud

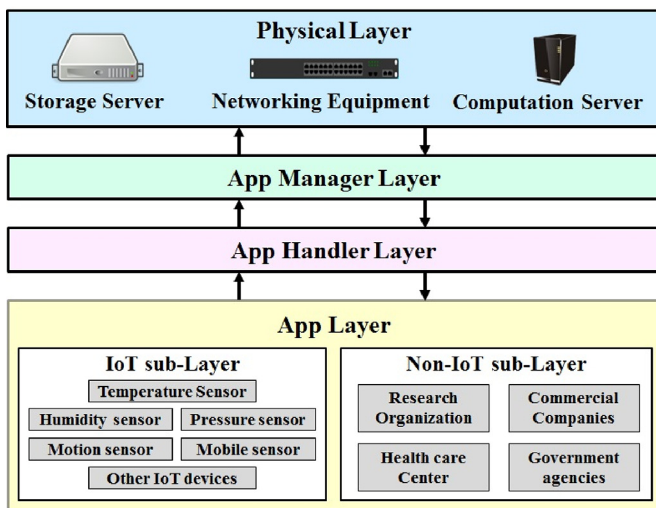


Fig. 2. Layer-wise architecture of the Cloud Framework.

consumer consumes services from CSP, Service Level Agreement (SLA) must be established between the consumer and provider. In general, the communication between cloud consumer and cloud provider is made by cloud broker. Cloud auditor is responsible for auditing and assuring the standard of the service provided by CSP.

Considering above discussed cloud framework, the layers of proposed architecture are *App layer, App handler layer, App manager layer* and *Physical layer* as depicted in Fig. 2. These four layers can be mapped into our proposed three layers of the cloud framework. App layer is responsible for generating realtime data or batch data to store in the cloud. The request to process those stored data are also generated from App layer. Hence, App layer can be mapped to the cloud consumer layer based on their nature of work. App handler layer, on the other hand is responsible for storing the data and processing the requests. This layer acts as a broker between the cloud and the actual IoT application. All the demands from the users and supply of the resources from the cloud provider can be negotiated through the App handler layer. Hence, the job of the App handler layer is similar to the job of the cloud broker. App handler layer is also responsible for analyzing the requests to determine the category of service required by the cloud consumer. For the sake of simplicity, we have considered 3 types of services such as SaaS, PaaS and IaaS and the type of services may vary according to the requirement of users in App layer. For example, for home security service, cloud service provider may be interested in providing Security as a Service, for research organization, cloud provider may provide Computation as a Service etc. Within the cloud provider, the huge amount of physical resources is managed by App manager layer through the resource virtualization, which is similar to the role of the service provider layer in cloud framework as shown in Fig. 1. The required amount of resources to process the incoming requests is further provided by the App manager layer based on the resource availability. In the next subsequent sections, we will elaborate about all layers in-details.

3.1. App layer

This layer is also considered as data source layer. Huge amount of realtime and batch data is being generated by different IoT devices based on different environment and send to the cloud to store for future analysis. For example realtime data generated from the smart traffic system, early earthquake detection and batch data generated from the sensors monitoring pollution levels in sea. We assume that the data generated by IoT devices can be used by multiple applications/ environments. Similarly, different sets of requests generated by other users, such as scientific research organization, belongs to this layer. Request could be generated by different mobile devices, and users of social media. The cloud environment provides enough resources to process and store those data on a realtime basis. The App layer is composed of two sub-layers: IoT sub-layer, Non-IoT sub-layer, as depicted in Fig. 2.

3.1.1. IoT sub-layer

IoT sub-layer is composed of all possible IoT devices, which can generate either realtime or batch data. The generated data can be structured, unstructured or semi-structured. Mostly, small amount of data are generated very frequently by different sensors. The data are generated so frequently that the aggregate amount of data becomes in thousands of petabytes of data. For example, a modern aircraft generates terabytes of data in every hour of flight. This huge amount of data is referred to as big data. Along with the volume, velocity and variety of the data, the data generated may contain unnecessary and ambiguous data. The value of the data needs to be extracted from the raw data for decision making that may need more computing capacity, which may not be fulfilled by the device itself. Furthermore, those processed data can be used by multiple applications such as the data from air pollution monitoring devices can be accessed by multiple government agencies.

3.1.2. Non-IoT sub-layer

Non-IoT sub-layer is responsible for generating requests to process those stored data. This sub-layer is composed of different components or users such as individual researcher, smart healthcare centers, government agencies, commercial companies etc. Let us consider an example to differentiate the IoT and non-IoT sub-layer. In a smart health care center, the patient can be fitted with different sensors those generate huge amount of data. On the other hand, doctors may send requests to the cloud to analyze and visualize the past health records of the patient. In this scenario, doctors, researchers data request and storage belongs to the non-IoT sub-layer, whereas the devices fitted in the patient's body come under the IoT sub-layer.

3.2. App handler layer

Cloud computing environment is designed to provide mainly two features; computation power and storage capabilities. But, when IoT environment is combined to cloud, different types of requests and data such as realtime and batch data are received by App handler layer, which resides in the cloud. Different types of requests such as request to analyze satellite data, medical image etc can be generated by users in App layer. On the other hand, realtime and batch data also generated by App layer. We assume that all the diversified data and requests need specific set of programs to store and to process. Keeping in mind the aforementioned requirements, a separated sub-layer is designed and incorporated to App handler layer, called App handler along with a specialized component is introduced within App handler layer, called as Request preprocessor as depicted in Fig. 3. Multiple App handlers can be implemented within App handler layer depending on the number of environments the cloud service provider is providing. Each App handler communicates with data station regulator that resides in Shared data station to access the data that are sharable among different applications. In the next sub-sections, we discuss about the App handler, Request preprocessor and Shared data station in details.

3.2.1. Request preprocessor

The requests are received by the cloud with the help of the sensor virtualization components, which reside in the App handler

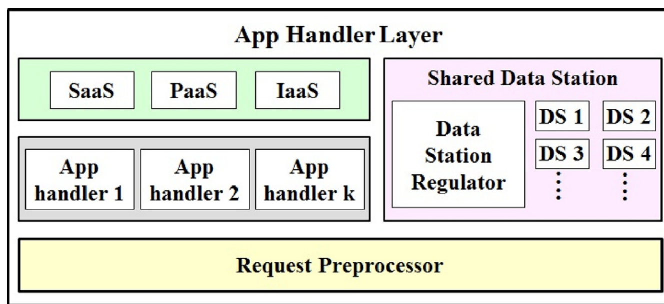


Fig. 3. Proposed architecture of the App Handler layer.

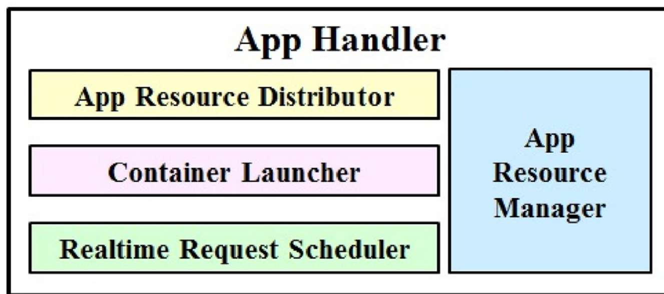


Fig. 4. Various components of App Handler.

layer. As discussed earlier, the received request can be for the storage or processing of the stored data. The data come into the cloud in the form of structured, unstructured or semi-structured format. Furthermore, the data could be from any types of sensors such as temperature sensors, GPS sensors, pressure sensors, humidity sensors etc. We assume that the incoming data may be ambiguous, which may contain unnecessary data such as sensor hardware information. The job of Request preprocessor is to forward the data/request in a predefine format to the respective App handler by removing unnecessary ambiguous data. In results, App handler components have no idea regarding the underlying sensors. App handler component will receive the request/data that is exactly what they need for further actions.

3.2.2. App handler

After requests are preprocessed by Request preprocessor, the processed requests are sent to the respective App handler sublayer. The processed request is then executed by this sub-layer. As we know, cloud provider may provide services to different smart applications or environments. For example, cloud provider may provide services to the government agency for smart cities application and at the same time this may provide services to a research organization in terms of high speed computing service for scientific data analysis. Dedicated App handler is implemented for each application and each component of the App handler is depicted in Fig. 4. The processed request forwarded by the sensor virtualization component is received by *Realtime request scheduler (RTRS)*. The processed request may come either from IoT devices or from an individual. Some requests require very small amount of computation such as visualizing past health record of a patient. On the other hand, some requests such as request that needs the specific platform and resource set require a dedicated virtual machine to process. Depending on the amount of resource required and the nature of the request, the incoming request can be categorized into SaaS, PaaS or IaaS. Hence, this is the job of the App handler to classify the incoming request into aforementioned categories. To handle those realtime requests coming to cloud, the App handler may face insufficient amount of resources and hence App handler need to send the resource demand frequently to App manager. This methodology helps cloud service provider to handle realtime data and requests.

Another important component of App handler is the *App resource distributor (ARD)*, which distributes the limited resources based on the requests. *App resources manager (ARM)* component manages the resources among various requests. Before scheduling, the scheduler informs about the resources requirement of requests to the ARM. ARM then verify whether the requirements can be fulfilled with current available virtual resources or not. If the aggregate amount of required resources by all the requests is less than that of the available virtual resources, App resource manager indicates ARD to do the resource distribution. Each small request is hosted by container launcher. Here, small request refers to the request, which is required by the SaaS environment. Hence, container launcher is placed to serve each SaaS request. For each SaaS request, a separate dedicated container is launched by container launcher to provide the secure and isolated environment.

Each separate dedicated container is responsible for hosting a single request. In case of data analysis and processing, the request may need huge amount of input data. Those required data are made available to the container that hosts the respective request. According to the proposed design framework, the container needs to communicate with the shared data station for the input data before sending the request for the input data. The Shared data station contains the data those are sharable among the multiple applications.

3.2.3. Shared data station

Shared data station is designed to keep track of the input data, which are required by the multiple applications. For example, the data generated from air pollution monitoring devices can be used by different research organization and government agencies. Multiple applications may share the data with other applications. In practical scenario, different sets of data can be requested by different applications. For example, Let us assume that three sets of data d_1, d_2, d_3 and two applications A_1, and A_2 are present in the App handler layer. Application A_1 may need data set d_1 and d_2, whereas application A_2 may need data set d_2 and d_3. Hence, the relationship between multiple applications and data sets is many-to-many. Data sets are enclosed within separate Data Stations (DSs). To provide seamless access among App handlers without any conflict, data station regulator is introduced. Data from the DSs are provided to the App handler via data station regulator.

The App handler layer has the complete control over the virtual resources to reconfigure and distribute among different requests. The required virtual resources are allocated by App manager layer, which works closely with the physical resource layer. Hence, the App handler layer needs to communicate with the App manager layer to acquire the required computation and storage power in form of the container and virtual machine. In the next subsection we will discus about the App manager in details.

3.3. App manager

App manager is responsible for managing the cloud resources and provide the physical resources in the form of virtual resources to different App handlers. App manager must also be capable of allocating resources to the App handler on realtime basis as App handler may send resource demand based on the incoming data and requests. The components those are involved in this process are *Physical resource distributor (PRD), Physical resource manager (PRM)*, and *Virtualization environment (VE)* as shown in Fig. 5. Here, we have incorporated the container technology to provide the host application. Container based virtualization environment can be provided to the cloud consumer as *Container as a Service* as some providers have started implementing to commercialize it.

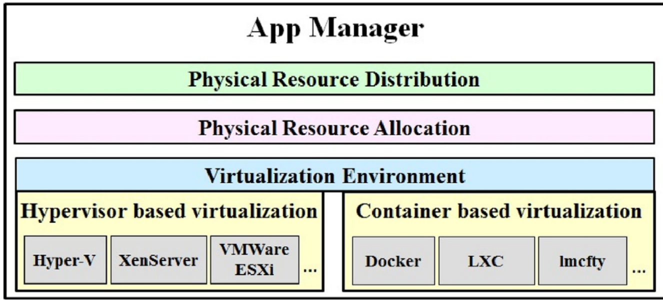


Fig. 5. Components of App Manager.

Physical Resource Distributor (PRD): Similar to the App resource distributor component in App handler, PRD distributes the available physical resources among different App handlers. We can conclude here that the available physical resources are distributed among requests in two different stages. In the first stage, the physical resources are distributed among different App handlers, which reside in App handler layer in the form of virtual resources. In the second stage, those virtual resources are distributed among the requests by the App resource distributor. The resource requirement of each App handler is received from the PRM. Based on the available physical resource, distribution is made. For fair distribution of resources among App handlers, the cloud provider needs to implement the resource distribution strategy. The resource distribution strategy will ensure the fairness and also ensures that best suitable physical servers are chosen to fulfill the resource demand. For example, the demand for resources may come from different geographical locations. Resource demand from region *A* should be fulfilled by the physical server present in the same or nearer region.

Physical resource manager (PRM): The resources that are distributed to App handlers, are managed by the PRM. The acceptance decision of the requirements of App handlers is made by the PRM. If aggregate demand from all App handler is more than the available physical resources at the cloud provider, the distributions are made based on the priority and the implemented distribution strategy. It is assumed that the priority of the App handlers is decided by the cloud service provider.

Virtualization environment (VE): From the proposed framework's design point of view, virtualization refers to virtual representation of physical resources. In standalone PCs, hard drive partition is one of the examples. This is implemented by using virtualization software. Operating system virtualization can run in multiple OSs at the same time in single computer system by using virtualization software. Different types of virtualization could be storage virtualization, network virtualization, server virtualization etc. In our proposed framework, the physical resources are provided to the App handler by the App manager in the form of virtual resources. Those virtual resources including memory, network, storage, etc are provided by encapsulating within a container or a virtual machine. For three kinds of services, such as SaaS, PaaS, IaaS, we have divided the virtualization environment into two types: hypervisor based and container based as depicted in Fig. 5. Hypervisor based virtualization is implemented by different software such as Hyper-V, XenServer, VMware ESXi etc. Similarly, container based virtualization is implemented by software such as Docker, LXC, lmcfty etc. The VE component provides the suitable environment based on the type of the services. For example, for SaaS request, the container based virtualization environment is provided. For PaaS and IaaS, hypervisor based virtualization environment is provided.

3.4. Physical layer

The underlined infrastructure for all layers to work properly is the set of available physical servers. We assume that the servers are equipped with different sets of resources and hence all servers are heterogeneous. Those underlined physical servers come under Physical layer. This layer is composed of hardware such as CPU, memory, storage etc. and communication hardware such as routers, switches, and power supply system, cooling system etc. The actual process execution is carried out in this layer.

4. Theoretical analysis

In this section, we present the mathematical representation of the discussed service framework. The cloud service provider needs to provide service to the users. The concern users are grouped based on the environment they are connected to. The services are provided to the App handler in terms of virtual machines and containers, which is then distributed to different users. App handlers are bridge between the users and cloud service provider. As discussed earlier, cloud service provider may need to host environment where data are generated and stored on realtime basis. For example, in VANET, lots of data related to the traffic congestion, location information need to be processed and stored on realtime basis and hence the resource demand may vary time to time. To facilitate such environment, we design the framework in such a way that App handlers can send request to the App manager for more resources at any instance of time. This will enable App handler to handle the realtime data and requests.

It is assumed that each App handler is associated with priority, which is assigned at the time of SLA establishment. Priority of an App handler can be defined as level of relative importance of the service given through the respective App handler. For example, suppose *App handler* 1 is designed to give health care related services and *App handler* 2 is designed to receive and process micro-blog related requests. In these cases, while competing for the limited computation resources, *App handler* 1 must be given higher priority over *App handler* 2. Let CSP has m levels of priorities termed as $P_1, P_2, P_3, \ldots, P_m$, where $P_1 > P_2 > P_3 > \ldots > P_m$. For each priority level, P_i^a is represented as the number of App handlers, where $1 \le i \le m$. App manager is providing service to k number of App handles. Different App handlers are termed as $A_1, A_2, A_3, \ldots, A_k$. Priority of an App handler can be represented as $A_i^p = P_j$, where $1 \le i \le k, \quad 1 \le j \le m$. For example, $A_3^p = P_2$ indicates that P_2 is the priority of App A_3. CSP is powered by a total of s number of servers represented by $S_1, S_2, S_3, \ldots, S_s$. Further, CSP provides r types of resources. Each resource type is represented as R_i, where $1 \le i \le r$. We assume that the servers are heterogeneous, which indicates that different servers have different sets of resources. Γ_i^j represents the maximum capacity of resource of type R_j of server S_i, whereas γ_i^j represents current capacity of resource of type R_j of server S_i, where $1 \le i \le s$ and $1 \le j \le r$. Total amount of resources of type R_j currently available at CSP can be calculated as sum of the current capacity of resource of type R_j with each server, which can be represented as

$$\psi^j = \sum_{i=1}^{s} \gamma_i^j, \qquad \text{where} \quad 1 \le j \le r \tag{1}$$

Similarly, the maximum capacity of resource of type R_j at CSP can be calculated as sum of maximum capacity of resource type R_j with each server S_i, which can be represented as

$$\Psi^j = \sum_{i=1}^{s} \Gamma_i^j \tag{2}$$

It is assumed that $0 \leq \Gamma_i^j < \Psi^j$, which indicates server S_i is equipped with negligible amount of resource of type R_j. Here, negligible amount refers to the amount of resource required only to run the server and that amount of resource is not intended for users. For example, storage server may be equipped with negligible amount of computation capacity. The resource demand of App handler is r-dimension resource vector and is represented as

$$\mathcal{D}_i^t = \langle R_1, R_2, R_3, \ldots, R_r \rangle \qquad 1 \leq i \leq k \tag{3}$$

Where the App handler index is represented by i, $1 \leq i \leq k$ and t represents the current time. Hence, $\mathcal{D}_i^t$ is the demanded resource vector of App handler A_i at time t. Resource demand vector does not contain the information regarding the number of users connected to the App handler as the number of users is dynamic. Similarly the allocated resources at time t is r-dimension resource vector and is represented as

$$\eta_i^t = \langle R_1, R_2, R_3, \ldots, R_r \rangle \qquad 1 \leq i \leq k \tag{4}$$

Where η_i^t represents the allocated resources of all resources types at time t to the App handler A_i. To avoid congestion in sending excess request for more resource demand, App handler needs to wait for specific time threshold δ_k and the threshold value δ_k can be decided by the App manager. The reason behind imposing of this condition is to avoid the situation, where App handler may send the resource demand before allocation of the previous resource demand. The boolean variable Ξ indicates, whether the demanded resource vector is fulfilled by the App manager at time t or not.

$$\Xi_i^t = \begin{cases} 1 & if \;\; \mathcal{D}_i^t \leq \eta_i \\ 0 & Otherwise \end{cases} \tag{5}$$

Finally, the objective function of the cloud service provider can be given as follows.

$$\max \sum_{i=1}^{k} \Xi_i^t$$

subject to,

$$\text{if } \; \Xi_i^t = 1, \quad \text{then } \mathcal{D}_i^t \leq \eta_i^t \qquad 1 \leq i \leq k$$

$$\sum_{i=1}^{k} \eta_i^t \leq \left\langle \Psi^1, \Psi^2, \Psi^3, \ldots, \Psi^r \right\rangle$$

$$\text{if } \; \Xi_i^t = 1, \quad \text{then } \Xi_j^t = 1 \qquad 1 \leq j \leq i$$

The meaning of this objective function is to provide services to maximum number of App handlers. The first constraint ensures that the boolean variable will become 1 if and only if the resource demand is fulfilled by the App manager. For all types of resources, the total allocated resource to each App handler must be less than the maximum capacity of the CSP, which is ensured by the second constraint. On the other hand, the third constraint indicates that the demand of App handler with higher level of priority must be fulfilled before the demand of App handler with lower level of priority.

Furthermore, in the proposed framework the IoT devices are considered to be heterogeneous and therefore the App handlers may receive the requests to process the realtime or non-realtime data. Let θ_k be the time interval between two consecutive requests received by the App handler A_k. It is assumed that the requests can be termed as realtime or non-realtime based on the value of θ_k is very small or very large, respectively. Using the virtualization technology, the incoming requests can be processed in parallel instead of sequentially. Let α_k be the number of requests, which can be executed in parallel taking the currently available physical resources by App handler A_k. Each App handler is engaged to receive and process unknown number of requests over a period of time. However, each request is assigned with a unique id as *rqid*. In order to reduce the waiting time of the request to 0, the execution time E_{rqid}^t of a request arrived at time t with unique id *rqid* can be derived as follows.

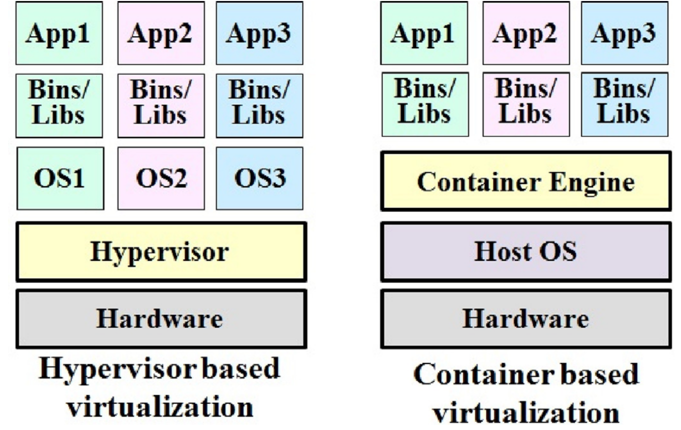


Fig. 6. Comparative views of Hypervisor based and Container based virtualization.

$$E_{rqid}^t = \alpha_k * \theta_k \tag{6}$$

In other words, the request received by App handler A_k at time t must have finished its execution within the time interval t through $t + (\alpha_k * \theta_k)$. Each time a request arrives at App handler A_k must wait certain time before its execution. This waiting time is caused due to new stacks. As discussed earlier, each App handler needs to wait δ_k units of time between two consecutive requests. Hence, a request that arrives at time t_1 has to wait for $\omega_1 = (\delta_k - t_1)$ units of time. It is also assumed that an App handler needs to wait for another small duration of time ω_2 after each resource is requested. In other words, ω_2 is the time required to distribute the resources to the App handler. Let ω_3 be the time required to schedule the request for the execution. Considering different delays caused by various components in our proposed framework SMFIC, the total waiting time of a request received by the App handler A_k at time t can be derived as follows.

$$\omega_{rqid}^t = (\delta_k - t) + \omega_2 + \omega_3 \tag{7}$$

5. Implementation of cloud platform

As shown in Fig. 6, virtualization is done in two ways. Firstly, hypervisor based virtualization and secondly, container based virtualization. Containers are very light weight virtual machines that are implemented at operating system level for running multiple processes in isolated environment. These

containers are best suited for hosting applications or software. Unlike hypervisor, containers share the kernel of host operating system. Hence, the container contains only the binaries of the application and the supporting libraries. The application inside the container accesses the hardware resource via the kernel of host machine. In our implementation we use Docker container technology (Docker - build, ship, and run any app, anywhere, 2016). On the contrary, in hypervisor based virtualization, conventional virtual machines are implemented at the hardware level with an operating system. Each virtual machine has separate operating system and kernel. All the virtual machines on a single physical machine are controlled by the virtual machine monitor. This virtual machine monitor is called hypervisor. Hence, unlike container, virtual machine contains the application, the supporting libraries and the underlined kernel. Hypervisor based virtualization is chosen to implement PaaS and SaaS model as depicted in Fig. 7.

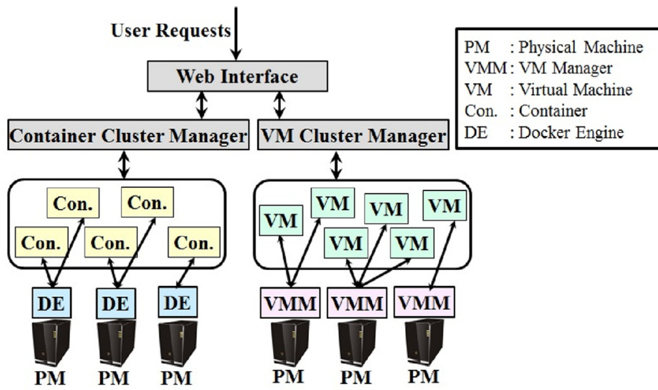


Fig. 7. Implementation architecture of SMFIC.

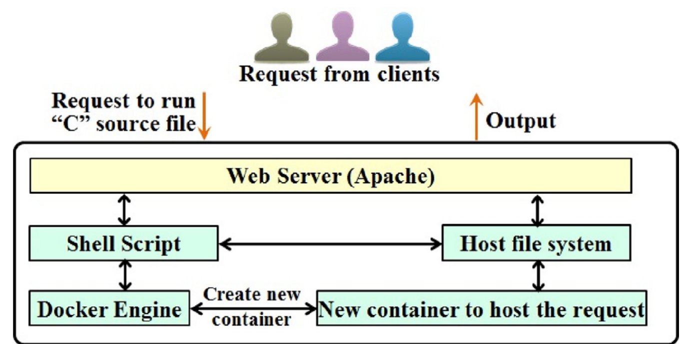


Fig. 8. Our implemented SaaS environment.

```
funlab@funlab-Docker-PC: ~
funlab@funlab-Docker-PC:~$ echo $cid2
fdb23e38ec33a52ed1a1eb1c59819f68
funlab@funlab-Docker-PC:~$ docker-machine create -d virtualbox --virtualbox-memo
ry 512 --swarm --swarm-discovery token://fdb23e38ec33a52ed1a1eb1c59819f68 worker
-00
```

Fig. 9. Command to create VMs using Docker-machine package.

```
funlab@funlab-Docker-PC: ~
funlab@funlab-Docker-PC:~$ echo $cid2
fdb23e38ec33a52ed1a1eb1c59819f68
funlab@funlab-Docker-PC:~$ docker run --rm swarm join --addr=45.55.48.62:2376 to
ken://^C
funlab@funlab-Docker-PC:~$  docker run --rm swarm join --addr=45.55.48.62:2376 t
oken://fdb23e38ec33a52ed1a1eb1c59819f68
```

Fig. 10. Command for joining a VM into a cluster.

The architecture of our Docker implementation is presented in Fig. 7. As shown in the figure, all the requests come to the cloud through the web interface, which can further be extended to the web application for more flexibility and more efficiency. Once the request is received through the web interface, cluster manager will process the request. As per our previous discussion, the cluster manager is of two types, container cluster manager and VM cluster manager. Container cluster manager is responsible for managing multiple clusters of containers. Similarly, VM cluster manager is responsible for managing multiple clusters of VMs. Furthermore, in a single physical machine, containers are placed above Docker engine. In other words, Docker engine in a single physical machine is responsible for creating, destroying and managing multiple containers. Each physical machine is setup with one Docker engine. The container created to serve a request is immediately destroyed after the request is processed. In our designed implementation architecture, we have considered to assign multiple containers to host one application or software. For example, to host a website, multiple containers can be created to host the file system, database system and other modules.

On the other hand, requests that need PaaS or IaaS are forwarded to VM cluster manager whose responsibility is to forward the request to a particular virtual machine based on VM selection strategy. Multiple virtual machines are created in each physical machine. VM manager in each physical machine creates, destroys and manages the virtual machines. The container or VM selection strategy is carried out in cluster manager.

In order to evaluate the performance of the proposed architecture, we have implemented the SaaS model onto our testbed. The testbed is composed of 20 systems; each configured with 8GB memory, 1TB of storage and is powered by Intel core i7-4790 CPU @ 3.6GHz CPU. Ubuntu 14.04 operating system is installed in all systems. For SaaS, we have developed a very basic application, named as *online C compiler*. Here, the user needs to upload the C source file through a web page. To receive the request, we have deployed Apache web server. In the background, for deployment of the container based virtualization technology, we have used Docker package. This allows us to create any number of containers through client console by executing following command, where the last argument is the image name of the application.

```
docker run <image name>
```

In our implementation, we have deployed the C environment upon Ubuntu base image, which can be obtained from the Docker repository using following command

```
docker pull Ubuntu:latest
```

After installing all required packages upon base Ubuntu image, we have pushed the image with the name *reachchinu/c-env* to Docker hub using following command:

```
docker push reachchinu/c-env
```

The received C source file by Apache web server is then forwarded to the Docker engine through shell script program using following codes.

```
docker run --rm
        --workdir =/home/funlab/c_env/$1
        -v /home/funlab/docker_app/cenv/$1:
        /home/c_env/$1
        reachchinu/c-env gcc 'prog.c'
        -o output
```

The Docker engine then creates a new container from the built image. The whole process is shown in Fig. 8. The newly created container interacts with the host file system to process C source file and the result is stored back to host file system, which then returned to the user through a web page. Docker engine creates and destroys the new container for each request. At a particular time, all the available containers process the incoming requests in an isolated manner.

We have further extended our experiment by forming a cluster of container by using the Docker-machine tool and Swarm package. The Docker-machine package allows us to create multiple virtual machines within a single physical machine and treat them as separate machines. The screen shots of the command shown in Fig. 9 can be used to create a virtual machine using Docker-machine tool. After creating multiple number of virtual machines, the available Swarm image from Docker hub is used to form a cluster of all virtual machines. For this, the screen shot of the used command is shown in Fig. 10.

The IP and port of each virtual machine is used in forming the cluster. The cluster formation is further extended by combining another virtual machine from the Digital Ocean using the command as shown in the screen shot in Fig. 11.

```
root@funlabuser-System-Product-Name: ~
root@funlabuser-System-Product-Name:~# docker-machine create -d digitalocean --digitalocea
n-access-token=eae52719fb5a505f3ee2ba80c5e3e3757477c050184f2bc92f71c51aca9e6744 swarm-DO
Creating SSH key...
Creating Digital Ocean droplet...
To see how to connect Docker to this machine, run: docker-machine env swarm-DO
root@funlabuser-System-Product-Name:~#
```

Fig. 11. Command used to create VM in Digital Ocean.

6. Performance evaluation

In this section, we evaluate the performance of our framework through simulation and implementation in Docker (Docker - build, ship, and run any app, anywhere, 2016) platform. Our simulation environment, corresponding simulation, experimental results and comparisons are described as follows.

6.1. Simulation setup

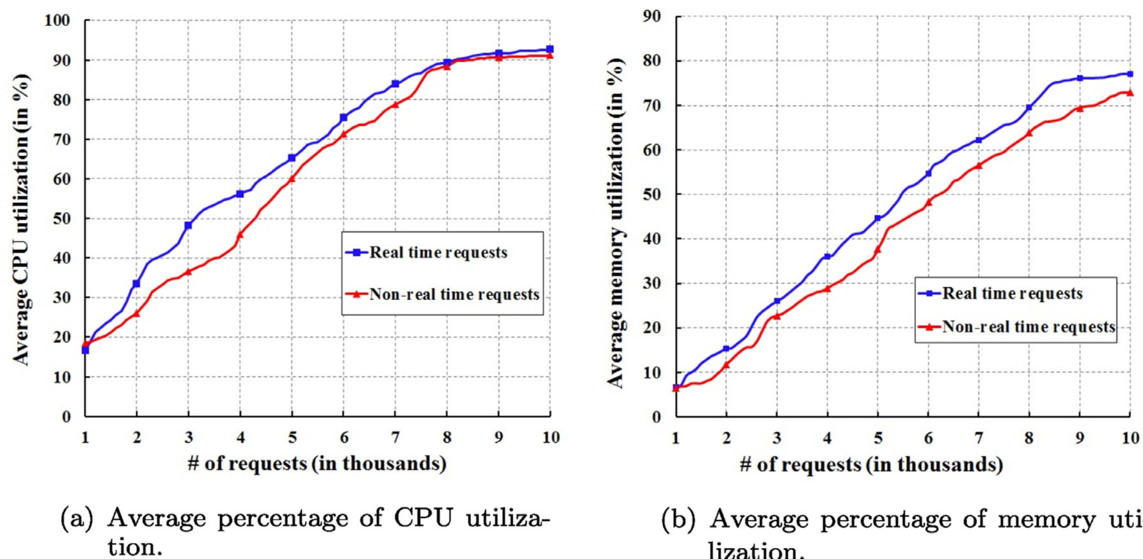
The simulation of our proposed framework is conducted using discrete event java based simulator, which is designed based on multi-server queueing system. In our simulation, it is assumed that the incoming requests follow Poisson distribution with mean ranging from 10 $\sim$ 100 requests per unit time. The service duration of each request follows random distribution and the resource demand of the requests is assigned randomly. The memory requirement of the request ranges from $2MB \sim 1024MB$ and the CPU requirement of each request ranges from 1 $\sim$ 2 cores. Similarly, the storage requirement of each request is taken to be $10MB \sim 10GB$. The requests are classified into two categories based on their resource demand. Firstly, the requests with very small resource demand, which are served by small containers. Secondly, the requests with huge amount of resource demand, which are executed by virtual machines.

In the simulation, the numbers of App handlers are set to be 15 and their priorities are assigned randomly. The priority values range from 1 $\sim$ 5. App handler with priority value 1 is treated as highest priority App handler, whereas App handler with priority value 5 is treated as the lowest priority App handler. The incoming requests are assigned to the App handlers randomly. The priority of a request is determined by the priority of the assigned App handler. The number of physical servers is set to be 50 throughout the simulation with random assignment of the resources. Servers are equipped with varied amount of memory ranging from $8GB \sim 16GB$ and the number of cores ranges from 8 $\sim$ 32 in each physical server. The amount of storage for each server is randomly assigned, which ranges from $500GB \sim 1TB$.

6.2. Simulation results

The average CPU and memory utilization with different numbers of real and non-realtime requests are simulated as shown in Fig. 12(a) and (b), respectively. In Fig. 12(a), it is observed that percentage of CPU utilization fluctuates in case of both real and non-realtime requests, when the number of requests is less. However, the CPU utilization by realtime requests is more than that of the batch requests when number of requests is more than 1000. It is also found that the CPU utilization by realtime and batch requests becomes saturated when number of requests are increased to more than 7500. The memory utilization by realtime and batch request is shown in Fig. 12(b), which shows the similar trend with CPU utilization. It is also observed that the memory utilization increases with increase in the number of requests, which gets saturated when the number of requests exceeds 8500.

In our simulation environment, both realtime and non-realtime requests are associated with certain amount of data, which need to be processed. In Fig. 13, the average processing time along vertical axis represents the average time required to process 50 TB, 100 TB and 500 TB of data, which are requested by 1000 $\sim$ 10, 000 number of requests. The time taken to process 50 TB, 100 TB and 500 TB of data is approximately 2 min, 4 min and 10 min, respectively, when the processed data are requested through 1000 requests. Similarly, the average time required to process 50 TB of data requested by 1000 requests is approximately 2 min, whereas the average time increases to approximately 6 min when the number of requests increases to 10, 000. This inclination in time is attributed due to various factors. For example, the physical resources need to be leased and released in form of VMs or containers as number of requests increases, which consume more time. It is found that the processing time increases slowly as the number of requests increases. On the other hand, considering the priority of the App handlers, average waiting time of respective requests is depicted in Fig. 14. It is found that the average waiting time of the requests with higher priority is less than the requests that belong to the lower priority. For example, the average waiting time of the requests that belong to the App handler with *Priority* 1



(a) Average percentage of CPU utilization.

(b) Average percentage of memory utilization.

Fig. 12. Average percentage of resource utilization: (a) CPU utilization (b) Memory utilization.

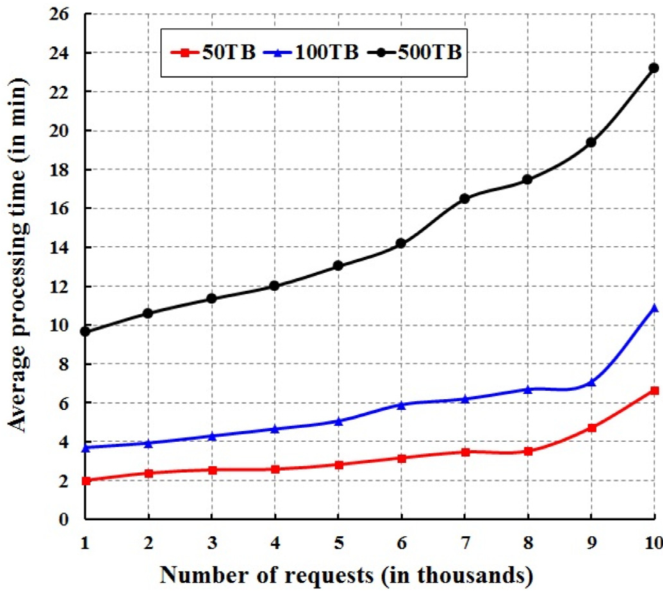


Fig. 13. Average processing time of the requests for different data sizes.

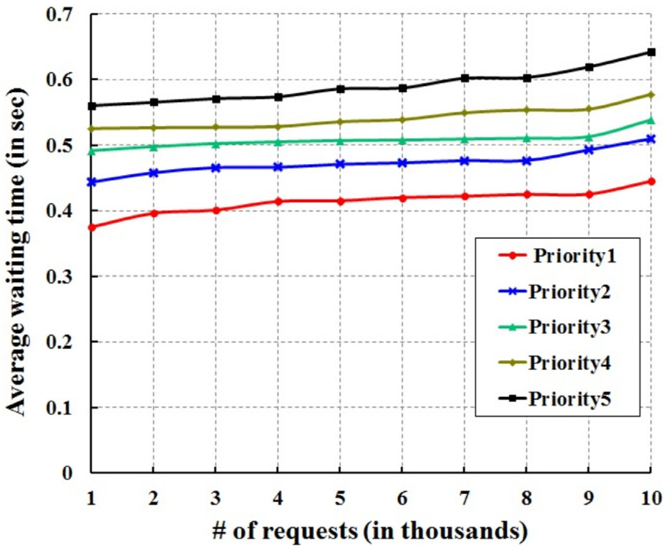


Fig. 14. Average waiting time of the requests with different priorities.

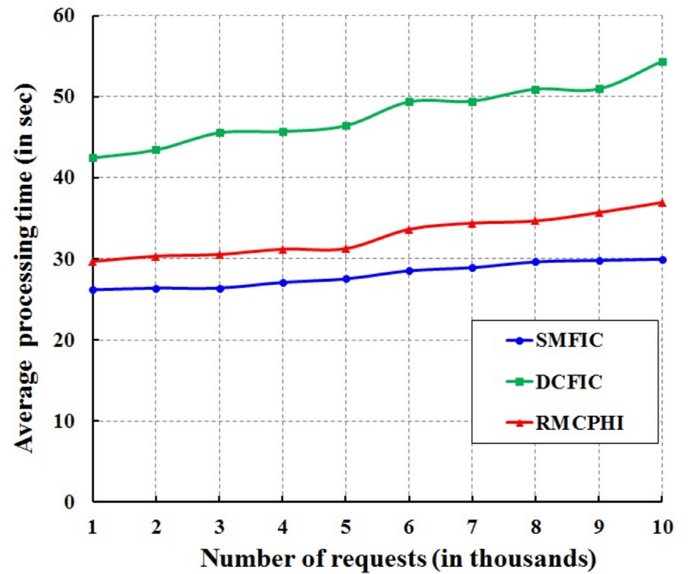


Fig. 15. Average processing time of requests.

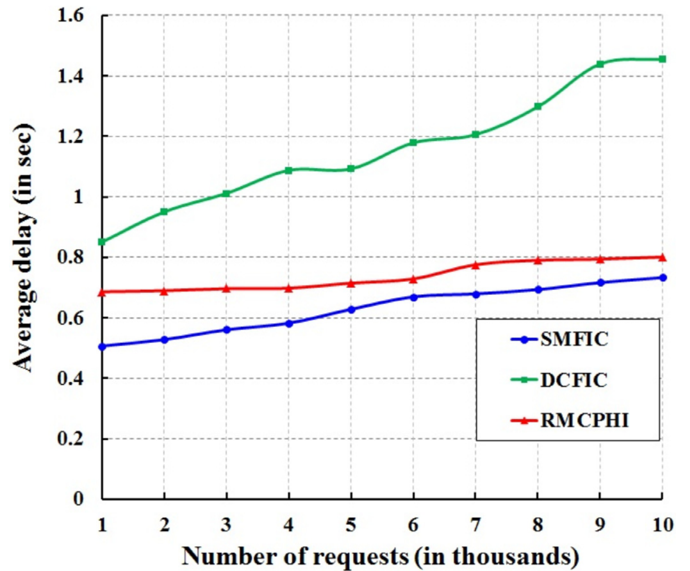


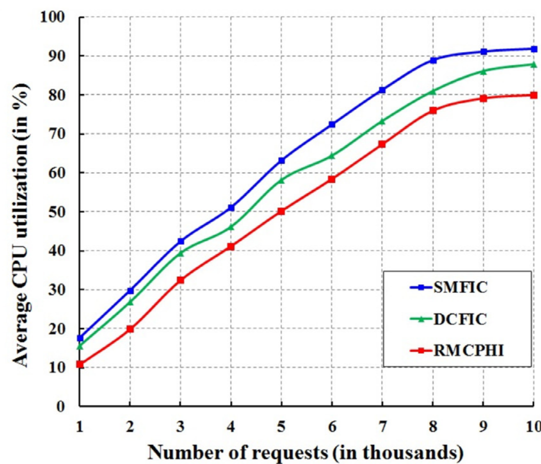
Fig. 16. Average delay (in sec).

ranges from 0.38 s to 0.44 s, whereas the average waiting time of the requests belong to the App handler with *Priority* 5 ranges from 0.55 sto 0.645 s. The resources are distributed to the App handler with highest priority followed by the App handler with lowest priority, which causes the increase in waiting time of the lowest priority App handlers.

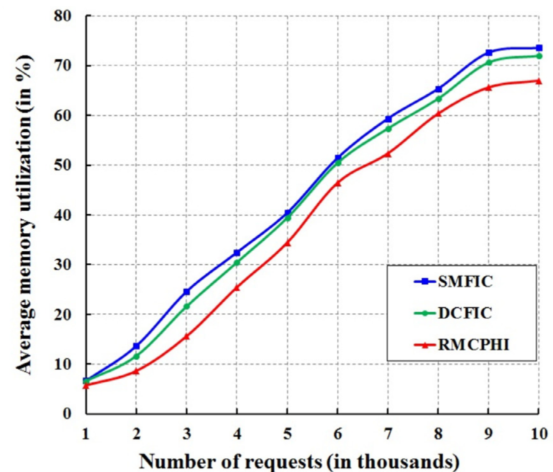
6.3. Comparisons

In order to show the technical contribution of our proposed SMFIC framework, it is compared with DCFIC (Khodadadi et al., 2015) and RMCPHI (Luo and Ren, 2016). In the proposed framework, the computational complexity is analogous to the processing time of the request, which can be define as the sum of delay and the time required to execute a request with certain amount of input data. The average processing time of the requests is simulated as shown in Fig. 15. The average processing time in SMFIC framework varies between 26 s through 30 s, whereas average processing time of the request in DCFIC framework ranges from 42 s through 54 s. In the simulation, the incoming requests are considered to be of limited lifetime. The average delay of the requests is simulated and compared with other frameworks as shown in Fig. 16. The average delay of a request is referred to the time spent in the waiting queue to be executed, which includes the delay caused by the resource unavailability and the delay due to new stacks. The results are obtained with varied number of requests ranging from 1000 to 10, 000. The average delay of the requests in DCFIC framework ranges from 0.8 s to 1.45 s, whereas the average delay of requests in SMFIC framework ranges from 0.52 s to 0.72 s. Container based virtualization technology enables us to provide the services in minimum delay with significantly less resource demand, which decreases the average delay in the SMFIC framework.

The average resource utilization for different numbers of requests is shown in Fig. 17(a) and (b). The resource utilization can be defined as the ratio of amount of allocated resources to the maximum available resources in each server. In both CPU and memory utilization, the proposed SMFIC framework outperforms



(a) Average percentage of CPU utilization.



(b) Average percentage of memory utilization.

Fig. 17. Comparison in terms of resource utilization: (a) CPU utilization (b) Memory utilization.

over DCFIC and RMCPHI framework. In SMFIC, we have considered that a physical server may suffer from resource fragmentation. In other words, the amount of aggregated resources allocated to different virtual machines is less than the total amount of resources equipped in the server. Hence, small amount of resources remain unused. Those unused resources can be allocated by creating new container for the requests with smaller resource demand, which maximizes the utilization of physical resources. As shown in Fig. 17(a) for CPU utilization, it is observed that the utilization increases with increase in the number of requests. However, the CPU utilization gets saturated when the number of requests crosses 8000. The same trend is observed in case of memory utilization of the physical servers as depicted in Fig. 17(b). The maximum memory utilization in the proposed framework is 73%, whereas the maximum memory utilization in RMCPHI is 67%.

As discussed earlier, in our implementation, the container-based virtualization is used along with hypervisor based virtualization technology. Docker is used to implement the container based virtualization. Although, the proposed framework considers SaaS, PaaS and IaaS cloud model as depicted in Fig. 2, for simplicity SaaS cloud model is implemented in our framework. Shell script, PHP scripting language and Java language are used to implement the functionality of different components. In our experiment, it is observed that the performance of Docker container is better than that of the virtual machines in terms of response time and processing time. In our implementation, response time is defined as the time required to start the VM or container and load the App/request onto the memory. Similarly, the execution time is considered to be the time required to start the VM/container and execute the App/request in the VM/container to get the desired output. Average waiting time and average execution time are obtained from multiple experiments.

As shown in Fig. 18, the average start-up time of a Docker container is faster than that of the virtual machine. Hence, as compared to the virtual machines, more number of containers can be assigned to a physical server without compromising the performance of the containers. As the containers need smaller amount of resources and share the same kernel among them, the execution time of the App running in the container is faster then the App running in the virtual machine, as depicted in Fig. 19. In our experiment, we have assigned a request to obtain the execution time, which will generate a file containing 1GB of random data

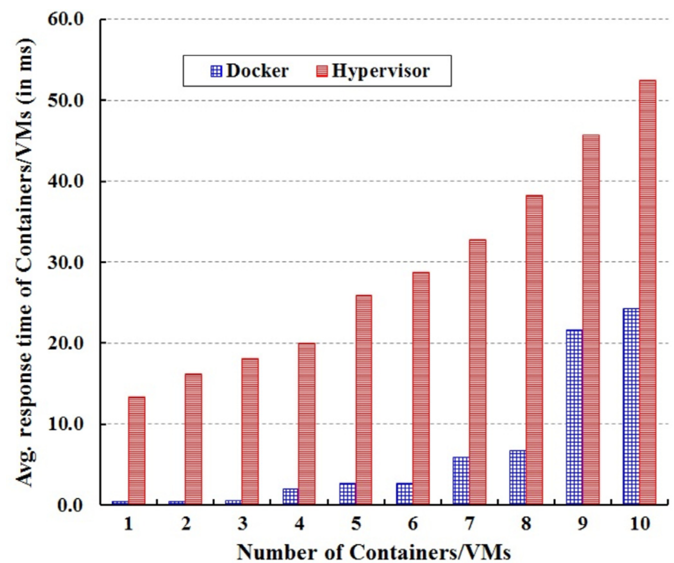


Fig. 18. Average response time of VMs/Containers.

and search a random word within the file using grep command. The execution time of the request is monitored. Each request is executed by one VM/container and each VM and container executes only one request. The execution time includes the time required to create the VM/container and the time required to execute the assigned request. From the experimental result, it is concluded that the performance of container is better than that of the virtual machines.

7. Conclusions and future works

The proposed cloud framework combines IoT and cloud environment to provide services to both IoT and non-IoT users. IoT part consists of various IoT and non-IoT devices, which are responsible for generating requests to process those stored data. The other part of the framework is Cloud, where data storage and process are carried out depending on the user requirement. Based on the requests made by the users, resources are given to process those requests. For IoT and non-IoT users/devices, process and storage of realtime

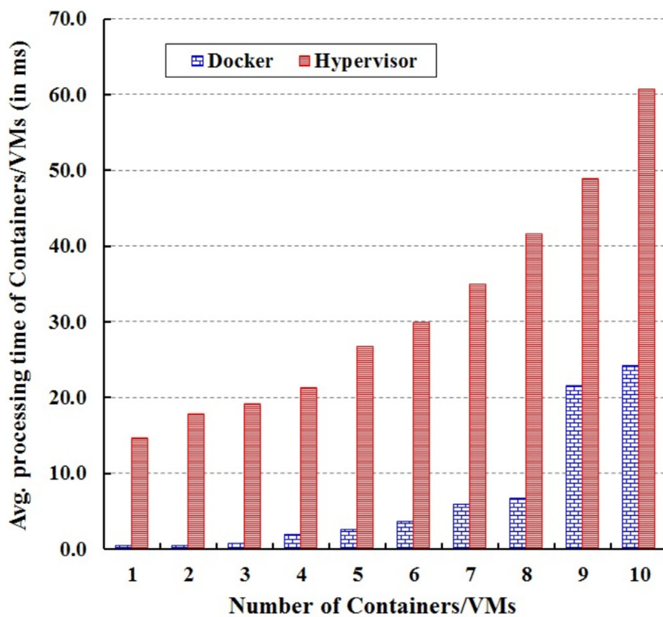


Fig. 19. Average processing time of Apps running in VMs/Containers.

data has been considered. To provide quality of service and utilize the available resources, the virtualization environment, which is the backbone of cloud environment, is divided into two parts. Firstly, container based virtualization for providing SaaS and secondly hypervisor based virtualization for providing PaaS and IaaS. In our work, the Docker Container based virtualization is implemented to provide SaaS. However, physical resource distribution algorithms should be designed to distribute the resources among App handlers in form of virtual machines and containers, which needs to be investigated and is considered as our future work. Besides, the future work of SMFIC framework includes the mechanism for App handlers to interact with the data station regulators and data stations efficiently with an objective to minimize the communication cost.

Acknowledgment

This work is supported by Ministry of Science and Technology (MOST), Taiwan under the grant number 104-2221-E-182-004.

References

Epc symposium,, 2003. inaugural epc executive symposium, URL: http://xml.coverpages.org/EPCSymposium200309.html.

Docker - build, ship, and run any app, anywhere,, 2016. URL: https://www.docker.com/.

Internet of things (iot) - cisco, 2016. URL: http://www.cisco.com/web/solutions/trends/iot/overview.html.

Itu-t recommendation database,, 2016. URL http://handle.itu.int/11.1002/1000/11559.

Atzori, L., Iera, A., Morabito, G., 2010. The internet of things: a survey. Comput. Netw. 54 (15), 2787–2805.

Bai, T., Rabara, S.A., 2015. Design and development of integrated, secured and intelligent architecture for internet of things and cloud computing. In: 3rd International Conference on Future Internet of Things and Cloud (FiCloud). IEEE, pp. 817–822.

Bitam, S., Mellouk, A., Zeadally, S., 2015. Vanet-cloud: a generic cloud computing model for vehicular ad hoc networks. IEEE Wireless Commun. 22 (1), 96–102.

Gurav, U., Shaikh, R., 2010. Virtualization: a key feature of cloud computing. In: Proceedings of the International Conference and Workshop on Emerging Trends in Technology. ACM, pp. 227–229.

He, W., Yan, G., Da Xu, L., 2014. Developing vehicular data cloud services in the iot environment. IEEE Trans. Ind. Inform. 10 (2), 1587–1595.

Jiang, L., Da Xu, L., Cai, H., Jiang, Z., Bu, F., Xu, B., 2014. An iot-oriented data storage framework in cloud computing platform. IEEE Trans. Ind. Inform. 10 (2), 1443–1451.

Joy, A.M., 2015. Performance comparison between linux containers and virtual machines. In: 2015 International Conference on Advances in Computer Engineering and Applications (ICACEA). IEEE, pp. 342–346.

Kelly, S.D.T., Suryadevara, N.K., Mukhopadhyay, S.C., 2013. Towards the implementation of iot for environmental condition monitoring in homes. IEEE Sensors J. 13 (10), 3846–3853.

Khodadadi, F., Calheiros, R.N., Buyya, R., 2015. A data-centric framework for development and deployment of internet of things applications in clouds. In: 2015 IEEE Tenth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP). IEEE, pp. 1–6.

Lee, E., Lee, E.-K., Gerla, M., Oh, S.Y., 2014. Vehicular cloud networking: architecture and design principles. IEEE Commun. Mag. 52 (2), 148–155.

Leukel, J., Kirn, S., Schlegel, T., 2011. Supply chain as a service: a cloud perspective on supply chain systems. IEEE Syst. J. 5 (1), 16–27.

Li, F., Vögler, M., Claeßens, M., Dustdar, S., 2013. Efficient and scalable iot service delivery on cloud. In: 2013 IEEE Sixth International Conference on Cloud Computing. IEEE, pp. 740–747.

Li, W., Kanso, A., Gherbi, A., 2015. Leveraging linux containers to achieve high availability for cloud services. In: 2015 IEEE International Conference on Cloud Engineering (IC2E). IEEE, pp. 76–83.

Liu, X., Li, Q., Lai, I.K.-W., 2013. A trust model for the adoption of cloud-based supply chain management systems: A conceptual framework. In: 2013 International Conference on Engineering, Management Science and Innovation (ICEMSI). IEEE, pp. 1–4.

Luo, S., Ren, B., 2016. The monitoring and managing application of cloud computing based on internet of things. Comput. Methods Prog. Biomed. 130, 154–161.

Mallissery, S., Manohara Pai, M., Ajam, N., Pai, R.M., Mouzna, J., 2015. Transport and traffic rule violation monitoring service in its: a secured vanet cloud application. In: 2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC). IEEE, pp. 213–218.

Qian, M., Hardjawana, W., Shi, J., Vucetic, B., 2015. Baseband processing units virtualization for cloud radio access networks. IEEE Wireless Commun. Lett. 4 (2), 189–192.

Schaffers, H., Komninos, N., Pallot, M., Trousse, B., Nilsson, M., Oliveira, A., 2011. Smart cities and the future internet: Towards cooperation frameworks for open innovation.. Future Internet Assembly 6656 (31), 431–446.

Suarez, J., Quevedo, J., Vidal, I., Corujo, D., Garcia-Reinoso, J., Aguiar, R.L., 2016. A secure iot management architecture based on information-centric networking. J. Netw. Comput. Appl. 63, 190–204.

Suciu, G., Suciu, V., Halunga, S., Fratu, O., 2015. Big data, internet of things and cloud convergence for e-health applications. In: New Contributions in Information Systems and Technologies. Springer, pp. 151–160.

Sun, E., Zhang, X., Li, Z., 2012. The internet of things (iot) and cloud computing (cc) based tailings dam monitoring and pre-alarm system in mines. Safety Sci. 50 (4), 811–815.

Vlacheas, P., Giaffreda, R., Stavroulaki, V., Kelaidonis, D., Foteinos, V., Poulios, G., Demestichas, P., Somov, A., Biswas, A.R., Moessner, K., 2013. Enabling smart cities through a cognitive management framework for the internet of things. IEEE Commun. Mag. 51 (6), 102–111.

Wang, L., Ranjan, R., 2015. Processing distributed internet of things data in clouds. IEEE Cloud Comput. 2 (1), 76–80.

Yinglei, B., Lei, W., 2011. Leveraging cloud computing to enhance supply chain management in automobile industry. In: 2011 International Conference on Business Computing and Global Informatization (BCGIN). IEEE, pp. 150–153.

Yu, J., Kim, M., Bang, H.-C., Bae, S.-H., Kim, S.-J., 2015. Iot as a applications: cloud-based building management systems for the internet of things. Multim. Tools Appl. 1–14. doi:10.1007/s11042-015-2785-0.

Zanella, A., Bui, N., Castellani, A., Vangelista, L., Zorzi, M., 2014. Internet of things for smart cities. IEEE Internet Things J. 1 (1), 22–32.

Chinmaya Kumar Dehury received BCA degree from Sambalpur University, India, in June 2009 and MCA degree from Biju Pattnaik University, India, in June 2013. Currently he is perusing the PhD degree in the department of Computer Science and Information Engineering, Chang Gung University, Taiwan. His research interests are in scheduling, resource management and fault tolerance problems of Cloud Computing.

Prasan Kumar Sahoo received the Master of Science in Mathematics from Utkal University, India in 1994, and Master of Technology degree in Computer Science from the Indian Institute of Technology (IIT), Kharagpur, India in 2000. He received the first PhD degree in Mathematics from Utkal University, India and second PhD degree in Computer Science and Information Engineering from National Central University, Taiwan in 2002 and 2009, respectively. He is currently an Associate Professor in the Computer Science and Information Engineering department and Director of International Affair Center of Chang Gung University, Taiwan. He was also Associate Professor in the department of Information Management, Vanung University, Taiwan and has worked in the Software Research Center of National Central University, Taiwan. His current research interests include analysis and prediction of IoT and Healthcare Big Data, scheduling and resource management problems in Cloud. He has served as the Program Committee Member of several IEEE, ACM and international conferences and Program Chair of ICCT 2010.