Implementation of Edge Computing with Containers

*

Selva Priya Selvarathinavel sese1123
Selva Priyanka Selvarathinavel sese9092
Poorwa Hirve pohi2375
Sudeep Galgali suga5267

***Abstract*—Although cloud computing as a centralized system has been established successfully, it still faces problems such as high latency and congestion in the network. Edge computing moves to a decentralized system which would reduce the problems faced by cloud computing. Our implementation focuses on the use of containers as a method for edge computing. Using significant experiments we have intuitively concluded that containerization is a justified method of deployment at the edge.**

I. Introduction

A. What is Edge Computing?

Edge computing offloads data and applications to the extremes of the network, replicating them across distributed servers closer to the user. By reducing the distance between the user and the server to essentially one hop, edge computing significantly reduces the time required for processing and responses and allows for a better user experience. [1]

B. What are Containers?

Containers are instances inside the kernel in the form of partitions, which is virtualization at the operating system level. They are lightweight and cause less overhead as they use the operating system's system calls and do not require emulation. Containers allow for different softwares to be run on the same backbone, thereby isolating the software. The size of container image which is smaller than Virtual Machine (VM) images makes it suitable to launch applications at the edge faster than VM based appliances. [1]

II. Motivation

The well known centralized system today is cloud computing. It has provided a successful architecture in order to do business and for every-day users. An example of cloud computing used are cloud services like Amazon Web Services (AWS) and Microsoft Azure. Amazon Elastic Compute Cloud (EC2) allows the creation of instances in any location in the world in order to perform cloud computing. However, cloud computing comes with some drawbacks. Despite load balancing algorithms, these systems face significant latencies when large amounts of data are required to travel between the user and the server. As users ultimately require fast and efficient systems, edge computing will play a large role in reducing the latencies of cloud computing. Edge computing focuses on data processing at the 'edge' of the network, a single hop away from the user. This will optimize performance and reduce the delays and latencies which occur in cloud computing architectures. This is essentially a decentralized system, which would reduce the network congestion and avoid single points of failures. For high performance with fast processing and quick responses, edge computing should be lightweight and computationally inexpensive. This can be achieved using containers.

III. History and Current Trends in Edge Computing

Edge computing is not a new term by any means, the advent of edge computing began in the 1990s when Akamai introduced the first content delivery network(CDN). This paradigm introduced nodes which cached images, video and audio in locations close to the user. Edge computing works similarly with one additional caveat that it provides computation power on these nodes. In 2006, a related concept cloud computing gathered momentum with the release of amazons EC2 service. In 2009 an article was published [2] which introduced the basis for modern edge computing. It described a two-layer architecture, the first layer is very close the modern day cloud architecture in that it is a centralized server which services requests from a client. The second layer termed cloudlet caches the state of the server and is located closer to the client geographically. This architecture is paramount in the development of distributed IoT applications and the present day edge computing.

In the current age there are many different proprietary edge computing solutions. AWSs Lambda@Edge [3] is a mixed system which runs an application in an aws instance located closest to the client. AWS Greengrass [4] is an IoT based solution which allows any local device to act as an aws compute instance. It uses lambda cloud in the backend to service requests at a rapid pace. It also allow local data filtering to process some data locally and transmit only a fraction of the data to the cloud for further processing. There is a similar service provided by Microsoft Azure as well, which is called the Azure IoT edge [5].

BOINC is a middleware software, developed by University of California Berkeley. It uses unused CPU and GPU cycles on a computer to do scientific computing. It follows a client-server architecture which communicate with each other to distribute and execute tasks and then return the results. It uses

remote procedure calls to implement most of its functionality [6].

Edge computing has trickled down to mobile devices as well. HTC introduced an application called Power to Give (built using BOINC) which allows mobile devices to be used as a computing platform. Each compound problem is divided into smaller problems which are distributed and executed across multiple devices. The results are pushed back to a centralized server [7].

With the advent of containers, Edge computing is moving towards containerized applications. Based on a study conducted by Century Link [8] it was observed that containerized applications had a reduced latency as compared to applications which ran on a Virtual Machine. A technology review of containers for Edge Computing was performed by Claus Pahl [1], in this paper he discusses the advantages and limitations of containers on the edge. There is a new enthusiasm in the tech world for utilizing the advanced container orchestration services such as Docker and Kubernates on the edge. This warrants a deeper look into the concept as there are a lot of questions that still needs to be answered.

IV. System Design

The system consists of a few servers running in the edge providing some service to the clients. These servers are containerized and verified for suitability with edge computing.

A. The Application

We have decided to implement a data processing application on the edge. The application works as follows.

- Clients send portions of data for processing
- Application is deployed on multiple containers on the edge
- Application processes the data and returns result to the client This application will run on multiple containers, multiple clients will use the services of this application.

B. Container Orchestration

We realized we would definitely need a container orchestration tool to manage instantiation and monitoring of containers. Two tools were discovered in the survey. Kubernetes [9] and Docker Swarm [10]. A brief survey of the two is in the tableI [11]. Considering the timeline of this project, it may be easier to use the Docker Swarm over Kubernetes. A consideration of GUI orchestration tools like Portainer [12] and Rancher [13] can also be made. A Docker Swarm Layout is as in Figure 1 [14]. The swarm orchestrates the containers running in the Agent Edge devices.

V. Edge Virtualization Plan - WIP in next section

As seen in Figure 2, a container engine runs on the host operating system in contrast to a virtual machine environment where multiple heavy weight guest OSs run on the host operating systems in which the applications are executed. Clients connect to the edge computer when they require access to its housed applications. Clients are not statically connected

TABLE I
Kubernetes and Docker Swarm Comparison

Kubernetes	Docker Swarm
Can overcome constraints of Docker and Docker API	Limited to the Docker APIs capabilities
Autoscaling based on factors such as CPU utilization	Services can be scaled manually
Do-it-yourself installation can be complex, but flexible. Deployment tools include kubeadm, kops, kargo, and others	Deployment is simpler and Swarm mode is included in Docker Engine
Uses a separate set of tools for management, including kubectl CLI	Integrates with Docker Compose and Docker CLI native Docker tools. Many of the Docker CLI commands will work with Swarm. Easier learning curve

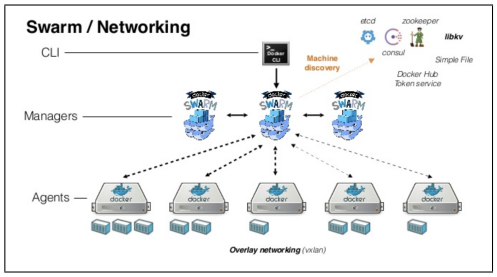

Fig. 1. Docker Swarm Layout

to the edge computer. Edges are typically laptop PCs that have low computing and storage capabilities compared to a data center. Typically in edge networks, clients send data collected through sensors provided depending on the application that has deployed the system. For every client, the edge creates a new container where data analytics and knowledge filtering can take place. Certain preliminary system issues that arise in this situation and how we propose to address them is given below.

A. Isolation of clients and limiting resources at the edge

One of the main responsibilities of the edge is to provide privacy for client nodes to perform their computations. This way, a particular client node will not be able to access the process of other client nodes and processes executing on the edge.This can be achieved through namespace isolation.Namespace isolation allows a group of processes to be separated. Docker containers provide a way to achieve this through managing namespaces. Docker uses process IDs and control groups to isolate certain processes within the container. Docker uses a Union File System as a storage unit for containers. With the help of process groups, the docker limits and monitors hardware resources used by all the containers in the edge without the running processes being aware of their limitations. All hardware components that have to be shared by the containers are combined into a format called

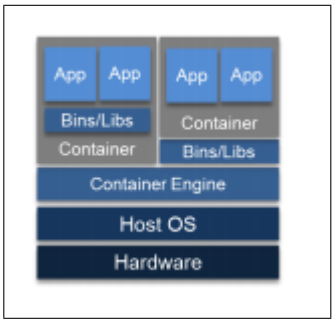

Fig. 2. Edge Virtualization Architecture [15]

libcontainer. [16] By default, there are no restriction of CPU or memory for each container. A container can use as much of a given resource as the host kernel scheduler allows. Docker provides ways to control how much CPU, memory or block IO a container can use with the help of restrictive options when the container is created.

B. Application Containerization

A repository contains multiple images than can be loaded or terminated on the container instances.An image contains the operating system that the container runs and a particular application enabled on it. Different user applications and platform components can be combined within a container too. The container API handles life cycle operations of the container like creating, deleting and distributing containers. It also handles running images on containers and different types of applications that can be deployed. The following image, Figure 3 shows different types of images that can be created and loaded on containers.

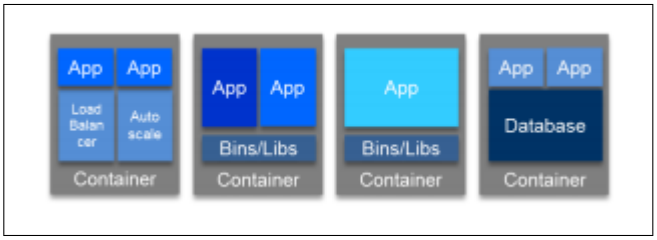

Fig. 3. Container Based Applications Architecture

C. Storage for clients at edge

Any application performed by the client at the edge must require a considerable amount of storage to perform computations.By default, a container is isolated from its neighbors. But without any access to data storage, they would have no useful value. Docker uses an overlay file system that stores any updated data to the root file system of the container. These changes are lost if the container is deleted. This happens when the client disconnects from the edge. Therefore, a container does not have persistent storage by default. However, docker provides two features that enable access to persistent storage namely docker volumes and data containers. [17] Data volumes are used to store data within the container and offers better performance for the application. It sits on the host file system. However, issues like an existing volume cannot be attached to a new or running container, which means a volume can end up orphaned. An alternative solution is to use a dedicated container to host a volume and to mount that volume space to other containers. This can be used as a method of sharing data within the containers at the same time. The last option remains to be to mount a local host directory into the container. This can be any directory other than the one under docker volumes folder. This is a better approach than the rest because the directory can be used by many containers at the same time and can already exist for new containers. [18]

D. Communication across clients' applications

Containers are self contained, isolated environments. But they are hardly useful if they cant talk to each other. This is true for our application if a client wants data transfer across an application executed by another client in its own container. Fortunately, the large number of connection options for containers enables a broad range of approaches, giving us the flexibility to choose an architecture that suits the needs of any application. The only way to achieve this is to make use of the Bridge interface in Docker. It works locally within one edge computer. Specific configurations to communicate that make use of the underlying bridge interface include Port Exposing, Port binding and Linking containers. An exposed port can be reached by other containers within the edge. If containers from other edges need to be able to access a container, port binding allows a container to bind a port of a container to a port of the edge that can be reached by other edges. The final approach is linking where two containers are linked and the docker will store the state of the connection via environment variables like IPs, ports used and hostnames involved in the linking. [19]

VI. Implementation

The works in this section are up and running.

A. The Application

We have decided to use a smart home application as an edge computing application. Obtaining devices and collecting sensor data is not feasible. Hence, we will use a simulator that simulates smart house data feed based on basic house specification. It takes: room layout, sensor layout, sensor specification and path to simulate as input; and outputs data. [20]

B. Inter-Container Communication techniques provided by docker

In most real world applications there is some form of communication between entities constituting the said application. Since we are running individual components of the application on docker containers, there is a need to identify the inter-container communication techniques provided by docker.

Docker provides two distinct means of communication between containers [21]. Network port mapping and linking

system. In the former approach we can map any network port inside a container within an ephemeral port range on the host machine. This allows any message sent to a particular port on the host machine to be forwarded to the respective container. We can also set a range of host ports to map to a single container. This allows the containers to send messages to other containers by referencing the respective ports on the host.

The second approach which is the legacy method is container linking. This system allows multiple Docker containers to be linked together thus allowing communication between them. When we create a link between two containers Docker creates a secure tunnel between them by adding each containers host name to each others etc/hosts file and also by sharing the environment variables between them [22].

There is a third method which is not provided natively by Docker but there is support for it. This is the shared namespaces method(ipc flag) [23]. Here two containers communicate by shared memory. There are two ways of implementing it, one way is to share the memory of the host machine with two or more containers, this method can be dangerous if one of the containers is compromised by a malicious user. The second approach is to create a dummy container and share its memory between the communicating containers. The latter provides a degree of safety against malicious attacks. This technique is beneficial in scientific computing and high performance computing where there are computers with incredibly high computing and processing power. This is less useful in an edge computing scenario where computing power is limited in the edge devices and it usually does not run many containers or instances of applications on the same node.

C. Application Architecture

Even though we have the means to communicate between containers by the aforementioned ways, it is primitive and leaves most of the burden of implementation on the developers. But fortunately there are plenty of open source data structures which can help us in implementing a fault-tolerant, reliable and highly available distributed application. Redis is one such technology which comes to mind. It provides hashes, lists, sets etc., has built-in replication, transaction processing, high availability and varying levels of on-disk persistence. With a very active development community there is support for distinctly different needs.

In this project we propose to utilize rediss queue data structure to facilitate inter-container communication. The application follows a producer-consumer model [24] in which there are one or more producers of information(these may be sensors deployed in the world). Then there are consumers which are our edge computing nodes running on individual containers which consume the messages/data provided by the producers and produce a result. The producers place items in the redis message queue(refer 4), the consumers are notified when there is a new item in the queue. Then one of the consumers picks up this item for further processing. There is a redundant processing queue which keeps track of the tasks which are currently being processed. In the event of a consumer module failing during work on a task, redis will be aware of a hanging task that has resided in the work queue for long, this task is then moved to the message queue so that another consumer can pick up the task.

With this particular approach we run into a single point of failure at the redis node. This problem can be dealt with by using redis replication feature. Redis uses a master-slave replication paradigm where the slave redis servers attempt to be exact copies of the master even during network issues(refer [25] for details on how it does this, it is pretty cool!). It uses asynchronous replication, the slaves asynchronously acknowledge the amount of data received periodically with the master. This is ideal for a completely distributed environment where failures are common. In our application we propose to use multiple replicated redis servers which are all connected to a redis sentinel service to provide active failover. [26]

In order to achieve high performance we need to have multiple redis masters which are connected via a load balancer. Any conventional load balancer would work in our case. The most commonly used industrial load balancer is HAproxy which provides both an open source as well as a proprietary version for different needs. This is currently not being looked in the project but it can be something that can be worked on in the future.

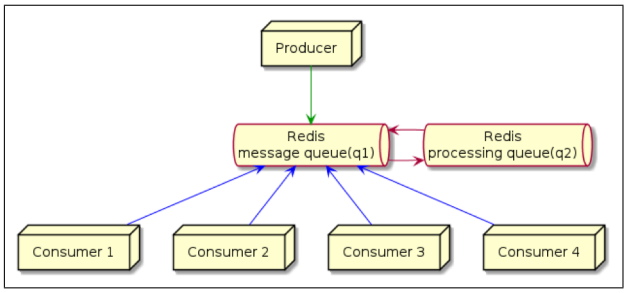

Fig. 4. Application Design

VII. Evaluation

Our evaluation must be designed to test the suitability of our solution to edge computing. Edge computing is characterized by the following principles that we will test.

- Memory usage
- Response Time

A. Space optimization

Any server needs space for storing the image (VM or container) and also needs memory to respond to client requests. Our evaluation will compare the space requirements of a VM based server and a container based server.

Table II records information of various image sizes. As we can see, the Ubuntu VM image is much larger than the Ubuntu container image. We can also get specific Python images for containers. The Alpine images are designed to be the smallest images. These images contain Python specific binaries ignoring other Linux binaries. A requirements file can be provided to install dependencies. As we can see from the

table, usage of this image reduces the application image size significantly. This is an important aspect that justifies usage of containerized applications in the edge.

TABLE II
IMAGE SIZES

Image	Size
Ubuntu Virtual Machine	1.02 GB
Ubuntu Container	113 MB
Ubuntu Container Python Application	432 MB
Alpine Python Container	69.5 MB
Alpine Python Container Application	85.2 MB

We also recorded the CPU and memory usage between the containerized application and a VM application. As we can see from Table III, there is no significant difference between the containerized and VM based application for the memory usage as the application runs. The application runs in the same way. One way to optimize this is to use the -m option in docker run to limit the memory to be used by the container. This too justifies usage of containers at the edge.

TABLE III
MEMORY AND CPU USAGE

Type	%Memory	%CPU
Containerized Application	1.13	55.01
VM Based Application	1.8	52

B. Response Time

The evaluation of the response time improvement of using edge over cloud services is performed in this section. First we tried running the servers in an Amazon ec2 instance and the client on one of our laptops simulating cloud service. Next we ran the servers on one of our laptops and the client on another laptop in the same network simulating edge service. We ran the application 10 times and the average application running time calculated at the client is in Table IV

TABLE IV
AVERAGE RUNTIME

Type	Runtime in seconds
Cloud Service	33.346
Edge Service	29.107

As we can see in Table IV, response time in edge computing is much faster in cloud computing as expected. Our application has maintained the edge computing capability.

REFERENCES

[1] C. Pahl and B. Lee, "Containers and Clusters for Edge Cloud Architectures – A Technology Review," *2015 3rd International Conference on Future Internet of Things and Cloud,* Rome, 2015, pp. 379-386. doi: 10.1109/FiCloud.2015.35

[2] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres & Nigel Davies, The Case for VM-Based Cloudlets in Mobile Computing, *IEEE Pervasive Computing* (Volume: 8, Issue: 4, Oct.-Dec. 2009)

[3] AWS Lambda edge, https://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html

[4] What is AWS Greengrass, https://docs.aws.amazon.com/greengrass/latest/developerguid is-gg.html

[5] Microsoft IoT Suite, https://azure.microsoft.com/en-us/suites/iot-suite/

[6] BOINC: A System for Public-Resource Computing and Storage, Proceedings of ACM, 2004

[7] HTC Power to Give, http://www.htc.com/us/go/power-to-give-faqs/

[8] Edge Computing Workloads, https://wp.cloudify.co/2017/10/22/edge-computing-workloads-vms-containers-bare-metal/

[9] Kubernetes https://kubernetes.io/

[10] DockerSwarm https://docs.docker.com/engine/swarm/

[11] KubernetesDocker https://platform9.com/blog/kubernetes-docker-swarm-compared/

[12] Portainer https://portainer.io

[13] Rancher http://rancher.com

[14] SwarmImage https://github.com/nkhare/container-orchestration/tree/master/swarm

[15] Containers and Clusters for Edge Cloud Architectures a Technology Review, Claus Pahl Irish Centre for Cloud Computing and Commerce IC4 and Lero, the Irish Software Research Centre Dublin City University Dublin 9, Ireland

[16] Evaluation of Docker as an Edge Computing Platform, Bukhary Ikhwan Ismail, Ehsan Mostajeran Goortani, Mohd Bazli Ab Karim, Wong Ming Tat, Sharipah Setapa, Jing Yuan Luke, Ong Hong Hoe *Advanced Computing Lab* MIMOS Berhad Kuala Lumpur, Malaysia

[17] How Storage Works in Dockers http://www.computerweekly.com/feature/Docker-storage-101-How-storage-works-in-Docker

[18] Persistent Storage in Containers https://www.networkcomputing.com/storage/docker-containers-and-persistent-storage-4-options/1320691891

[19] Connecting Docker Containers https://deis.com/blog/2016/connecting-docker-containers-1/

[20] https://github.com/So-Cool/SHgen

[21] https://docs.docker.com/network/links/#communication-across-links

[22] https://docs.docker.com/network/#network-drivers

[23] https://dzone.com/articles/docker-in-action-the-shared-memory-namespace

[24] https://blog.rapid7.com/2016/05/04/queuing-tasks-with-redis/

[25] https://redis.io/topics/replication

[26] https://redis.io/topics/sentinel

[27] A Performance Evaluation of Lightweight Approaches to Virtualization, CLOUD COMPUTING 2017 : *The Eighth International Conference on Cloud Computing, GRIDs, and Virtualization*