High Availability of Distributed Application based on Container migration

Nagthej Manangi Ravindrarao and Qicang Qiu
Computer Engineering Department
San Jose State University (SJSU)
San Jose, CA, USA. Email: {Nagthej.manangiravindrarao, Qicang.qiu@sjsu.edu

***Abstract*— The availability of services across distributed platform has been implemented using virtualization over the period. However, using Virtual Machines as Hypervisors results in significant limitations such as high resource overhead and system dependencies. This becomes a critical issue when it comes to deploying the application into production. The state-of-the-art container based platform called Docker, supersedes the conventional VM based approach. All the requirements to run the software application are packaged into isolated containers. However, the container based platform also has some limitations such as dynamic resource allocation, migrating containers from one host to another and providing High Availability.**

Our project aims to overcome these issues by using a Model driven based approach. We run web application in container and replicate it to demonstrate Load balancing. We join multiple hosts to form a cluster by enabling the swarm mode in multi-manager configuration. At the backend, we run a discovery service called consul in container to achieve the Failover mechanism between hosts. We use cgroups to provide CPU, RAM resources dynamically. This ensures the service is always up and running making the application Highly Available.

For efficient container migration, application checkpoint and Restore mechanism is needed which is provided by CRIU. We checkpoint the running web application and restore it to other hosts. By this way, the users experience minimum downtime during migration. We also extend the application to Nvidia-DIGITS container by leveraging GPU of NVIDIA GeForce 840M and implement object detection/classification.

***Index Terms*— Container migration, Cgroups CRIU, Docker, Docker Swarm, GeForce GPU, High availability, Model based, Nvidia-DIGITS, Virtualization, Virtual machine**

I. INTRODUCTION

With the advent of Docker release as an Open source on March 2013, it has brought radical changes in deployment of applications in software industry. Before the release of Docker, we used to achieve High availability [12] of applications using traditional hypervisor based platform. But due to increased overhead, resource intensity and system dependencies, it became increasingly difficult to manage in the production systems. Also, this approach requires more investment for building and shipping applications across multiple hosts

Container based virtualization [16] has emerged into an efficient way to build, deploy and ship applications across multiple hosts. Containers have the property of low resource consumption [11], Ease of portability and is Lightweight. Unlike VMs [15], containers do not envelope a complete operating system; only the required libraries and settings are needed to deploy the software. The applications are shipped to Docker Hub which is a cloud repository for sharing and migrating workflows to other cloud platforms.

Machine vision [1] has developed for 15 years since the beginning. This can be defined as an application system. Its functional features are gradually improved and developed with the development of industrial demands. At present, the total visual market of the whole world is about 60~70 million US dollars, which is increasing according to the growth rate of 8.8% per year.

In this paper, we are dockerize a web application which is implemented on Nvidia GeForce 840 M. We checkpoint the running web application using CRIU which is implemented natively in Docker. The dump files will be in the form of images which contains information of current host file descriptors, cgroups, mounts, and TCP connections. We then copy all these files to multiple hosts remotely. Then we restore the container from that specific checkpoint. The application immediately boots up and continuous its job from where it was left off. We further provide an abstract of how to achieve container migration for DIGITS (Deep learning GPU training system) container. The use of GPU is to leverage the GPU ability with the help of Nvidia-Docker wrapper tool.

The main challenges to be addressed are:

A. *Dynamic resource allocation:*

Docker does not provide solution to increase or decrease the amount of resources of containers (eg CPU, RAM usage, Network etc..) during runtime.

B. *Compatibility issues with Docker and Nvidia-Docker*

The containers are platform agnostic and hardware agnostic. It views the kernel as a black box. Hence there is need to externally provide GPU access to inside of containers.

C. *Live migration of containers:*

To save the state of the containers before downtime and to restore the session without experiencing any switch over.

D. *High Availability of applications*

We must ensure there is smooth and faster failover so that users do not experience any application downtime.

E. *Training model selection*

There are so many training models to choose from, and each model has its own advantages. Different applications need not only to select different models, but also to add specific layers to the model.

F. *DIGITS 5 Implementation*

Loading the dataset inside the container using Nvidia-docker volumes. Digits are available in different versions of the Linux operating system, when loading data, there will be errors that cannot be identified by the path.

II. RELATED WORK

A. Docker-Container

Many efforts have been put before to migrate a running container to remote host. But these methods lack one or the other feature to successfully migrate. Some of the work related to "cloud computing which address resource management at runtime with hypervisor based solutions [5] [6] [7] [8]". However, their work did not verify the application in production environment.

Many works have been proposed to checkpoint a container, but when checkpointed, the application is stopped and the users would experience a serious downtime [10].

B. OpenCVs

Open Source Computer Vision Library is an open source computer vision and machine learning software library. OpenCV is lightweight and highly efficient, composed of a series of C function and a small amount of C++, Python, Ruby, and MATLAB language interface, implements many common algorithms of image processing and computer vision. It is committed to real-world real-time applications, through the preparation of the optimized C code has brought considerable improvement on its execution speed, and can be purchased through Intel IPP high performance multimedia library (Integrated Performance Primitives) to get faster processing speed. The performance of the OpenCV is compared with the current mainstream visual function libraries as shown in figure 1.

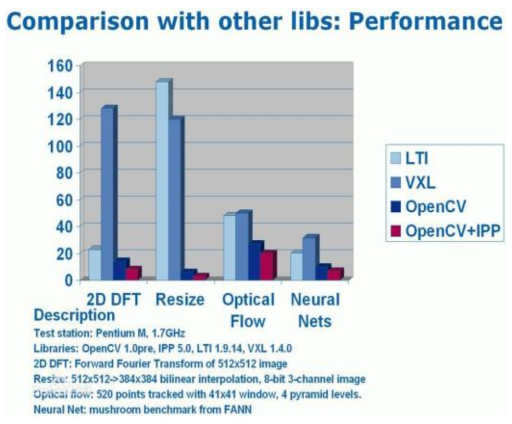


Figure 1: Comparison with other libraries

III. SYSTEM OVERVIEW (INNOVATIVE DESIGN)

Docker uses a client-server based architecture. The Docker client communicates with the Docker daemon- which builds, runs and distributes containers. The Docker client and daemon can run on same host, or we can connect Docker client to a remote Docker daemon machines. The client and daemon exchanges information using REST API, which is on top of UNIX sockets or on a Network interface.

First, we present the overview of solution architecture, then we divide the major modules into sub-modules to explain in detail. The overview is presented in figure 2.

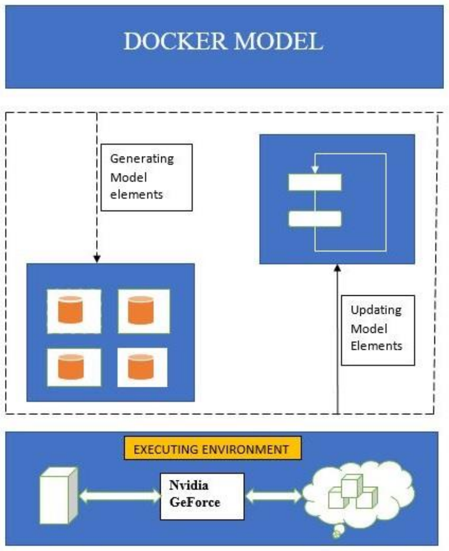
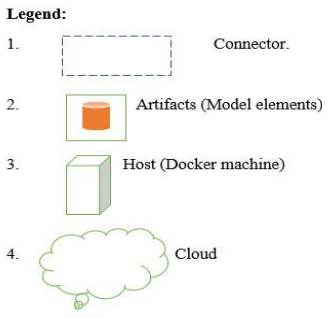


Fig. 2. Architecture Overview



The architecture is mainly divided into 3 parts:

1. Docker model
2. Connector
3. Executing environment

The architecture presents Docker model [9] as an expressive for containers. The connector provides the linkage between the Docker model and the Executing environment. The Docker engine present in Docker model can create N number of containers.

It also operates seamlessly to make updates for Docker model [11] elements according to changes that is taking place in Executing environment. Finally, the generated artifacts are executed in the Executing environment.

The executing environment consists of Multiple physical of Virtual hosts or even hosts that are running on cloud. The Containers deployed on host equipped with GeForce 840M is managed by a single host that communicates with this host bidirectionally. All the nodes joined to the machine form a Docker swarm in a multi-manager configuration and they are assigned their status (Manager, worker etc.)

IV. TECHNICAL DISCUSSION

Our key objective is to achieve Container migration. The container migration concept was first Introduced around 2008 and is still not yet fully accomplished work. This concept was developed by Russian company called CRIU – checkpoint and Restore in userspace. They introduced live migration of Linux containers and eventually the concept was developed to other platforms. Docker incorporated this CRIU tool to their software natively. This tool can only be accessed in Docker when the experimental mode is enabled.

Our objective is to provide dynamic resource allocation, Container migration, and providing High Availability.

A. Dynamic Resource Allocation

To provide CPU, memory, disk, network resources dynamically at runtime we use Docker connector. The connector manages the Docker resources by modifying their corresponding cgroups. cgroups are very important tool for managing resources used up by containers.

B. Live migration

Migrating a container essentially means transferring all the files inside a container to another container within the host or to another host. The files include data volumes, file descriptors, cgroups, kernel configurations, mounts, TCP connections and other libraries and dependencies which are very important for a container to restart.

The CRIU provides ability to migrate live Linux containers and play a role that maintains Quality of service by appropriate replication. The algorithm to checkpoint and restore is represented in figure 3.

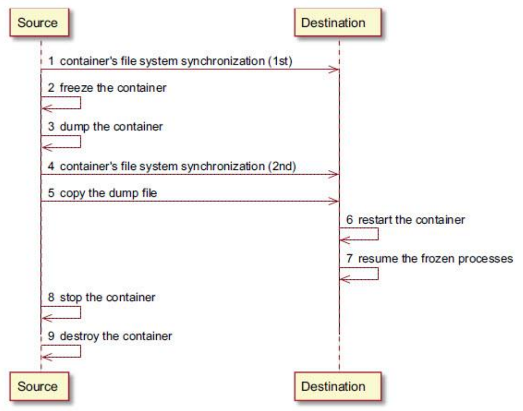


Fig. 3: Algorithm flowchart for container migration

The benefit of this implementation is that the migration process can rollback to source and it can resume the suspended container on source when there is failure due to any reason such as network dysconnectivity etc. Providing dedicated services will depend on time consumed in stages 3 to 5.

Two optimizations are provided:

1. Lazy migration
2. Iterative migration

1. Lazy migration

The objective is to decrease the time consumed for file synchronization at stage 4 of figure 3. The advantage of this approach is container can be resumed on destination machine without waiting for full memory copy from source.

The module for lazy migration is shown below:

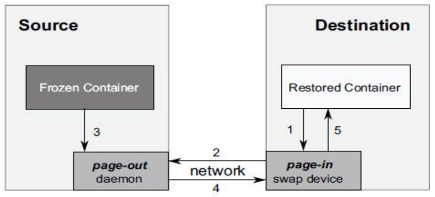


Fig. 4. Lazy Migration

The fundamental idea of lazy migration is to transfer only a minor subset of memory pages to destination host, and resume container on destination. Then pull the leftover pages from source on demand.

When the page miss occurs, the container sends a request to page-in swap device who then redirects the request to page-out daemon resides on source to pull missing page.

By this way, the requested page is transferred and is loaded into memory on destination as given in steps from 1 to 5.

Finally, when container is idle for a given time slot, a last swap out actions is forced and remaining pages are transferred from source to destination.

The migration can also be done Iteratively by transferring memory pages and file systems to synchronize before freezing the container.

This will reduce the amount of data needed to transmit after freezing the container.

But iterative migration is not suitable for this web application and Nvidia-DIGITS. Hence, we have not discussed it in detail.

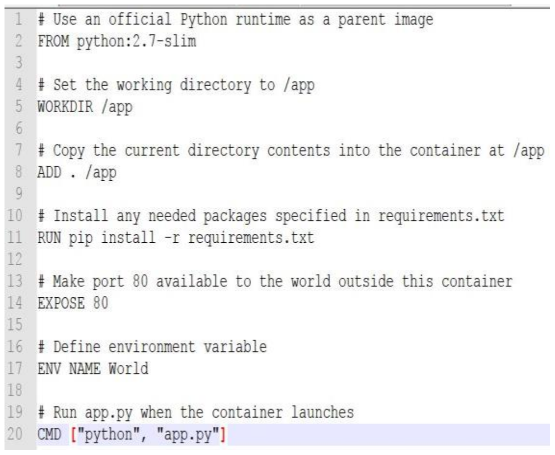
Container Migration procedure

First a Dockerfile is written using the syntax given by docker.inc. Dockerfile shown in figure 6. From that file, we build a base image using “docker build” command and run a container using “docker run” command on one host with IP: 10.0.0.221.

Now to Initiate swarm mode. We must make one host as manager. We make the GeForce enable host as manager by “Docker swarm init” command. This issues a token number to join to the swarm as workers.

Now using this token, the other two hosts are joined as workers. This forms a cluster where the command can be executed on any host.

When commands are executed on worker nodes, behind the scenes it just routes those commands to manager to get it executed.



```
# Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
ADD . /app

# Install any needed packages specified in requirements.txt
RUN pip install -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

Fig 5: Dockerfile

Now we deploy our services to scale up the containers. We run docker-compose.yml file. Finally, we run the app.py file where we get “hello world” message in localhost. The screenshots are provided in section 5 under experiment and evaluation. We start the discovery service consul in the backend. Consul along with etcd or zookeeper can be used. All 3 are compatible with swarm mode.

Now all the 3 hosts will send heartbeats to the consul where the host with IP: 10.0.0.221 gets elected as leader.

The 3 nodes along with Discovery service is illustrated in figure 6.

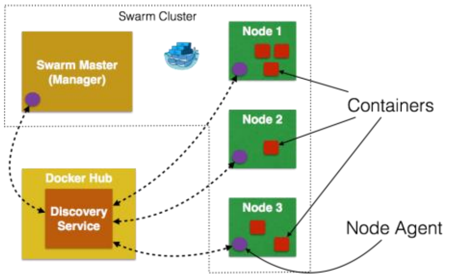


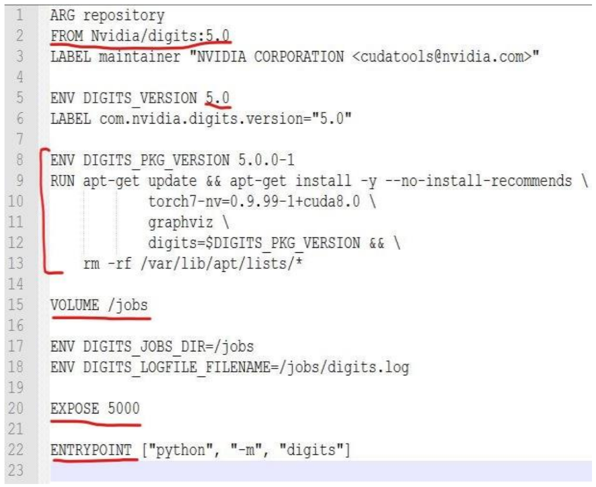
Figure 6: Nodes with consul discovery service (source: google images)

Now CRIU is used to create checkpoint and restore it. We create checkpoint and restore it to all 3 nodes in the cluster. To copy the dump files to remote host we used rsync Linux command since it is more powerful than scp to transfer files. We can also use shared file systems between multiple hosts such as NFS.

If the manager host is down for some reason (including maintenance of server), the manager node cannot send heartbeat to consul. The consul will know that host is down. Now the other 2 hosts compete for the key to get elected as a leader. The IP with the healthiest node gets elected as leader and we can observe the transfer of leadership. This way if the host is down, the leadership is transferred to backup host ensuring service is always up and running.

The DIGITS Application part is explained below:

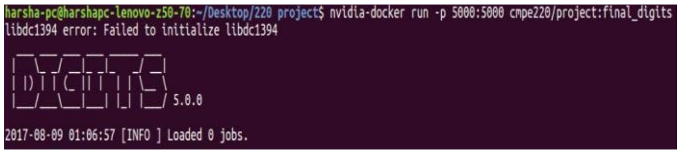
Everything starts with a Dockerfile. We again write the Dockerfile with all the necessary software tools required to run DIGITS in container. Dockerfile illustrated in figure 7



```
ARG repository
FROM Nvidia/digits:5.0
LABEL maintainer "NVIDIA CORPORATION <cudatools@nvidia.com>"

ENV DIGITS_VERSION 5.0
LABEL com.nvidia.digits.version="5.0"

ENV DIGITS_PKG_VERSION 5.0.0-1
RUN apt-get update && apt-get install -y --no-install-recommends \
            torch7-nv=0.9.99-1+cuda8.0 \
            graphviz \
            digits=$DIGITS_PKG_VERSION && \
    rm -rf /var/lib/apt/lists/*

VOLUME /jobs

ENV DIGITS_JOBS_DIR=/jobs
ENV DIGITS_LOGFILE_FILENAME=/jobs/digits.log

EXPOSE 5000

ENTRYPOINT ["python", "-m", "digits"]
```

Figure 7: Dockerfile for Nvidia-DIGITS

Before running the container, we also write docker-compose.yml file to scale up the replicas. Now we use nvidia-docker wrapper to run DIGITS container as shown in figure 8



```
harsha-pc@harshapc-lenovo-z50-70:~/Desktop/220 project$ nvidia-docker run -p 5000:5000 cmpe220/project:final_digits
libdc1394 error: Failed to initialize libdc1394

  ___ ___ ___ ___ _____ ___
 |   \_ _/ __|_ _|_   _/ __|
 | |) | | (_ || |  | | \__ \
 |___/___\___|___| |_| |___/ 5.0.0

2017-08-09 01:06:57 [INFO ] Loaded 0 jobs.
```

Figure 8: DIGITS container using nvidia-docker

A. Application

Our dataset consists of around 150 photographs which are IPhone, IPad and the Samsung Galaxy (50 each). We use Digits to implement cell phone type identification. 75% of pictures were used for learning. We then tested the model on four pictures which were found on Google images. The model did well on all of them getting all four classifications correct.

B. AlexNet

AlexNet is the name of a convolutional neural network, originally written CUDA to run with GPU support, which competed in the ImageNet Large Scale Visual Recognition Challenge in 2012. The network achieved a top-5 error of 15.3%, more than 10.8 percentage points ahead of the runner up. AlexNet was designed by the SuperVision group, consisting of Alex Krizhevsky, Geoffrey Hinton, and Ilya Sutskever. [2]

The one that started it all (Though some may say that Yann LeCun's paper in 1998 was the real pioneering publication). This paper, titled "ImageNet Classification with Deep Convolutional Networks", has been cited a total of 6,184 times and is widely regarded as one of the most influential publications in the field. Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton created a "large, deep convolutional neural network" that was used to win the 2012 ILSVRC (ImageNet Large-Scale Visual Recognition Challenge). For those that aren't familiar, this competition can be thought of as the annual Olympics of computer vision, where teams from across the world compete to see who has the best computer vision model for tasks such as classification, localization, detection, and more. 2012 marked the first year where a CNN was used to achieve a top 5 test error rate of 15.4% (Top 5 error is the rate at which, given an image, the model does not output the correct label with its top 5 predictions). The next best entry achieved an error of 26.2%, which was an astounding improvement that pretty much shocked the computer vision community. Safe to say, CNNs became household names in the competition from then on out. [3][4]

In the paper, the group discussed the architecture of the network (which was called AlexNet). They used a relatively simple layout, compared to modern architectures. The network was made up of 5 conv layers, max-pooling layers, dropout layers, and 3 fully connected layers. The network they designed was used for classification with 1000 possible categories. [3][4]

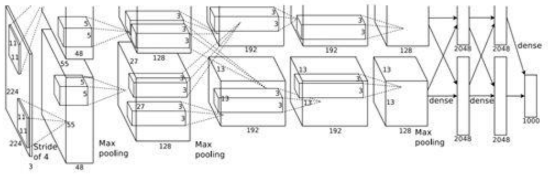


Fig. 9. Network architecture

V. EXPERIMENT AND EVALUATION

First, we need to configure and Install necessary tools to build the applications inside container. These steps are carried out to all 3 hosts shown in experiment setup in figure 10.

- Enabled the Host to use NVIDIA Geforce GPU unit instead of Intel Hd graphics family.
- Installed Docker-ce 17.06.01 version successfully
- Installed Nvidia-docker version 1.0.1 successfully, this wraps up docker commands and automatically provides container the GPU access to run apps (DIGITS in our case).
- Pulled and installed CUDA development libraries

Fig. 10. Experimental setup (All 3 hosts are x84-64-bit architecture running Docker on Ubuntu 16.04 LTS)

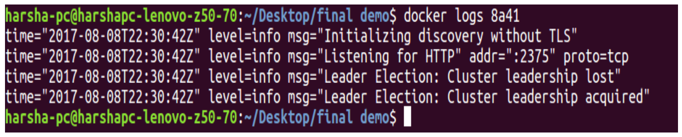
There are 3 hosts with all the necessary software and tools installed.

- Manager node - The GeForce enabled GPU is the host in the middle with IP: 10.0.0.221.
- Worker node - The host in left side has IP: 10.0.0.30
- Worker node - The host on right side of figure has an IP: 10.0.0.183

VI. RESULTS

We experimented the container migration of web application and obtained promising results. The results are shown as screenshots of the output we obtained.

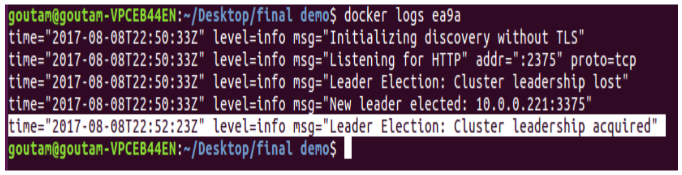
1. When the web application is running on the manager node, we can see the logs of that container. Here in figure 11 we see that initially the manager node with IP: 10.0.0.221 has the cluster leadership.



```
harsha-pc@harshapc-lenovo-z50-70:~/Desktop/final demo$ docker logs 8a41
time="2017-08-08T22:30:42Z" level=info msg="Initializing discovery without TLS"
time="2017-08-08T22:30:42Z" level=info msg="Listening for HTTP" addr=":2375" proto=tcp
time="2017-08-08T22:30:42Z" level=info msg="Leader Election: Cluster leadership lost"
time="2017-08-08T22:30:42Z" level=info msg="Leader Election: Cluster leadership acquired"
harsha-pc@harshapc-lenovo-z50-70:~/Desktop/final demo$
```

Fig. 11: 10.0.0.221 has cluster leadership

2. Now to test the migration of container, we manually stopped the container at manager node 10.0.0.221. The consul discovered this change and immediately transferred the leadership to 10.0.0.30 and a different container is started in that node as shown in figure 12.

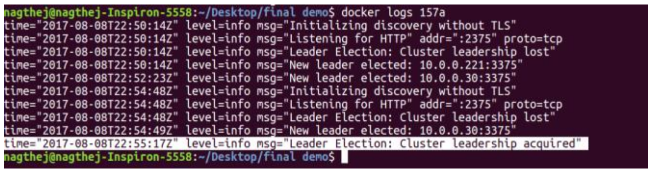


```
goutam@goutam-VPCEB44EN:~/Desktop/final demo$ docker logs ea9a
time="2017-08-08T22:50:33Z" level=info msg="Initializing discovery without TLS"
time="2017-08-08T22:50:33Z" level=info msg="Listening for HTTP" addr=":2375" proto=tcp
time="2017-08-08T22:50:33Z" level=info msg="Leader Election: Cluster leadership lost"
time="2017-08-08T22:50:33Z" level=info msg="New leader elected: 10.0.0.221:3375"
time="2017-08-08T22:52:23Z" level=info msg="Leader Election: Cluster leadership acquired"
goutam@goutam-VPCEB44EN:~/Desktop/final demo$
```

Fig. 12: Leadership transfer from 10.0.0.221 to 10.0.0.30

From the figure 12, we can also observe that initially it discovered the leader as 10.0.0.221:3375 and later when that IP got stopped, it switched over to 10.0.0.30

3. To further demonstrate migration process, we stopped the host with IP: 10.0.0.30 and observed the leadership transferred to 10.0.0.183 as shown in figure 13.



```
nagthej@nagthej-Inspiron-5558:~/Desktop/final demo$ docker logs 157a
time="2017-08-08T22:50:14Z" level=info msg="Initializing discovery without TLS"
time="2017-08-08T22:50:14Z" level=info msg="Listening for HTTP" addr=":2375" proto=tcp
time="2017-08-08T22:50:14Z" level=info msg="Leader Election: Cluster leadership lost"
time="2017-08-08T22:50:14Z" level=info msg="New leader elected: 10.0.0.221:3375"
time="2017-08-08T22:52:23Z" level=info msg="New leader elected: 10.0.0.30:3375"
time="2017-08-08T22:54:48Z" level=info msg="Initializing discovery without TLS"
time="2017-08-08T22:54:48Z" level=info msg="Listening for HTTP" addr=":2375" proto=tcp
time="2017-08-08T22:54:48Z" level=info msg="Leader Election: Cluster leadership lost"
time="2017-08-08T22:54:49Z" level=info msg="New leader elected: 10.0.0.30:3375"
time="2017-08-08T22:55:17Z" level=info msg="Leader Election: Cluster leadership acquired"
nagthej@nagthej-Inspiron-5558:~/Desktop/final demo$
```

Fig. 13: Acquired leadership from 10.0.0.30

The fundamental reason while both the IP 10.0.0.30 and 10.0.0.183 work is that nodes in a cluster participate in an ingress routing mesh. This ensures that the specific IP with that port is always reserved for that application only.

DIGITS Application results (Cell phone detection)

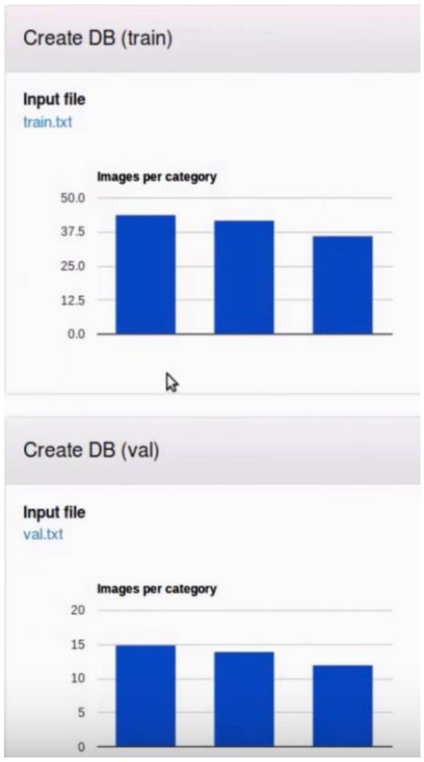


Fig. 14: DB training

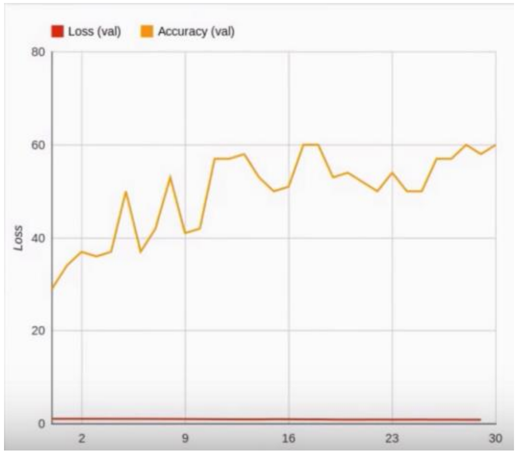


Fig. 14: Dataset of cellphone detection

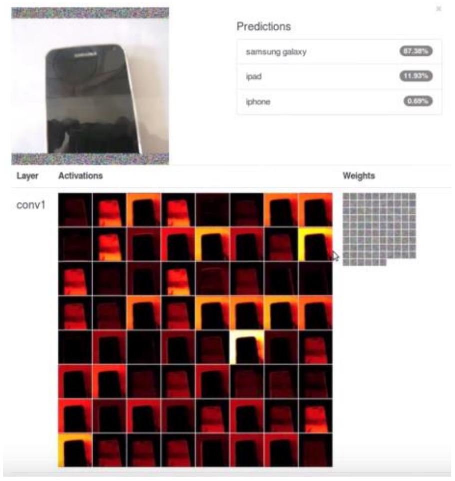


Fig. 15: Output results of Samsung galaxy detection

Figure 14 represents the data set. The column diagram above shows the training set of the 3 devices, and the column diagram below shows the validation set of the 3 devices. Figure 4 shows the final recognition results. Figure 5 is one of the additional pictures we have tested. The results on the right show that the phone in the picture are 87.38% likely to be Samsung galaxy which is correct.

VII. CONCLUSION

The Model driven approach presented in this paper successfully migrated a Distributed running web application from one host to another. The Docker model, connector and the executing environment were all synchronized. Results obtained by migrating containers were promising. A full stack development of an application has been achieved. Our approach of using Discovery service consul is more reliable compared to existing solutions. Through the use of clustering solution provided by swarm combined with Discovery service, High Availability of web Application is ensured.

We also experimented with simple image classification using DIGITS. We choose 150 pictures of 3 different mobile phones, each with 50 pictures. 75% of the images were used as training sets, and 25% of the remaining pictures were used to test the accuracy of the results. In addition, we downloaded the pictures of these three mobile phones randomly from the Internet and classified them by DIGITS. The model did well on all of them getting all four classifications correct. An attempt and Huge progress has been made to migrate the DIGITS application.

The future work will be focused on migrating DIGITS container between hosts. This includes debugging of the dump log file to fix errors concerning mount issues, non-regular mapping and providing TCP connections.

VIII. TEAM

1. Nagthej Manangi Ravindrarao

Nagthej is currently pursuing Masters in Computer engineering at San jose State university, CA, USA. He received his bachelor's degree in Global Academy of Technology, Bangalore, India. He has been worked in the field of embedded programming, Internet of things for 1 year. His research interests include Virtualization, Cloud computing and IoT.

2. Qicang Qiu

Qicang is currently pursuing Masters in Computer engineering at San jose state university, CA, USA. He completed his bachelor's degree in China. He worked as an intern in china on Computer vision applications. His research area includes Computer vision, and Image processing.

REFERENCES

[1] W. Bledsoe. Man Machine Facial Recognition. Technical Report, PRI:22, Panoramic Research Inc., Palo Alto, USA, 1966

[2] D. Gershgorn. (n.d.). The data that transformed AI research—and possibly the world. Retrieved from https://qz.com/1034972/the-data-that-changed-the-direction-of-ai-research-and-possibly-the-world/

[3] Deshpande, A. (n.d.). The 9 Deep Learning Papers You Need to Know About. Retrieved from https://adeshpande3.github.io/adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html

[4] A. Krizhevsky, I. Sutskever, & G. E. Hinton, (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097-1105)

[5] J. C´aceres, L. M. Vaquero, L. Rodero-Merino, ´A. Polo, and J. J. Hierro. Service scalability over the cloud. In *Handbook of Cloud Computing*, pages 357–377. Springer, 2010.

[6] B. P. Rimal, A. Jukan, D. Katsaros, and Y. Goeleven. Architectural requirements for cloud computing systems: an enterprise cloud approach. *Journal of Grid Computing*, 9(1):3–26, 2011.

[7] H. N. Van, F. D. Tran, and J. M. Menaud. Performance and power management for cloud infrastructures. In *2010 IEEE 3rd International Conference on Cloud Computing (CLOUD)*, pages 329–336. IEEE, July 2010.

[8] L. M. Vaquero, L. Rodero-Merino, and R. Buyya. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45–52, 2011.

[9] J. B´ezivin. On the Unification Power of Models. Software and System Modeling (SoSym), 4(2):171–188, 2005.

[10] M. Jammal, H. Hawilo, A. Kanso, A. Shami, Mitigating the Risk of Cloud Services Downtime Using Live Migration and High Availability-Aware Placement, 2016 IEEE 8th International Conference on Cloud Computing Technology and Science.

[11] F. Paraiso, S. Challita, Y. Al-Dhuraibi, Philippe Merle. Model-Driven Management of Docker Containers. IEEE. 9th IEEE International Conference on Cloud Computing(CLOUD), Jun 2016, San Francisco, United States. IEEE CLOUD pp.718 - 725, 2016.

[12] V. Stankovski, S. Taherizadeh, I. Taylor, A. Jones, C. Mastroianni, B. Becker, H. Suhartanto. Towards an Environment Supporting Resilience, High-Availability, Reproducibility and Reliability for Cloud Applications.

[13] Official Docker documentation: https://docs.docker.com/

[14] Official Nvidia Forum: https://devtalk.nvidia.com/default/topic/1000224/jetson-tx2/docker-on-the-tx2/1

[15] W. Li and A. Kanso, Comparing Containers versus Virtual Machines for Achieving High Availability, 2015 IEEE International Conference on Cloud Engineering.

[16] R. MORABITO Ericsson Research, Ericsson AB, Jorvas 02420, Finland, Virtualization on Internet of Things Edge Devices With Container Technologies: A Performance Evaluation, 2017 IEEE. Translations and content mining are permitted for academic research only.

[17] Self citation: This report partial content was submitted to canvas by myself "Nagthej Manangi Ravindrarao" and "Qicang Qiu" as Project design.