



Performance Evaluation of WebRTC Server On Different Container Technologies

Kubernetes and Docker Swarm

Naga Vyshnavi Kukkapalli

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Master of Science in Electrical Engineering with emphasis on Telecommunication Systems. The thesis is equivalent to 20 weeks of full-time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:

Author(s):

Naga Vyshnavi Kukkapalli

E-mail:

nagavyshnavik@gmail.com

naku16@student.bth.se

University advisor:

Dr. Dragos Ilie

Assistant Professor of Telecommunication Systems

Department of Computer Science and Engineering

University Examiner:

Prof. Kurt Tutschku

Department of Computer Science and Engineering

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

ABSTRACT

Background:

Cloud computing technology has come a long way with various technological advancements in the past few years. It has been accelerated with the evolution of various virtualization technologies. Currently almost every social platform and small-scale applications look towards cloud to deploy their services successfully and provide maximum satisfaction to their end-user. Thus, virtualizing their services becomes utmost important to deploy and develop their applications. This alone emphasizes the importance of Docker containers in the development world. Docker containers right now are playing a very important role in the field of cloud computing. Since Multimedia plays a huge role in our day to day lives and most people crave for faster and efficient responses, it is essential to develop our applications with better Real time communication capabilities. Thus, we are determining which container orchestration tool serves best for Real time communication applications.

A multimedia application is developed and deployed using WebRTC based Kurento media server and the performance of the server is measured when the application is deployed. We have chosen Kubernetes and Docker Swarm as container platforms for this thesis. The Servers and Clients are virtualized and metrics such as CPU Utilization, Network Traffic, Container overhead, Memory Utilization are measured. These metrics provide the performance overhead in different scenarios for each orchestration technology. This will be helpful to analyze and understand the effect of Kurento server on these technologies. Thus, the results are expected to determine which orchestration technology serves best for RTC applications.

Objectives: The objectives of this project are:

- To implement WebRTC based Kurento server in a container orchestrated environment.
- To extract performance metrics such as Network Traffic, CPU and Memory Utilization while server is running.
- To compare WebRTC based Kurento server in Kubernetes and Docker Swarm.

Method: Kubernetes and Docker Swarm environments are setup and then docker images with video conferencing application(One-to-One call and One-to-Many call) using Kurento media server is deployed in them. Once either of the applications is running, experiments are performed for analyzing performance metrics like CPU Utilization, Memory Utilization, Network Traffic and overhead using monitoring tool, Prometheus. Along with Kubernetes and Docker Swarm, Kurento server is also deployed on a stand-alone container to estimate the performance overhead. Later, statistical analysis(ANOVA and differences of Standard error) is done over these metrics and conclusions are drawn.

Results: Based on the performed experiments and the extracted metrics, for One-to-One call application, Kubernetes showed better resource utilization for CPU and Network Traffic while it consumed more memory over Docker Swarm. Similar behaviour is observed for One-to-Many application. When application is scaled, the percent of resource utilization increase in Kubernetes is higher when compared to Docker Swarm, but overall resource utilization of Kubernetes is much lower than that of Docker Swarm.

Conclusions: WebRTC based Kurento media server is investigated in Kubernetes and Docker Swarm. From the detailed analysis there is significant overhead in Docker Swarm than in Kubernetes for CPU Utilization and Network Traffic. For Memory Utilization, this is opposite. Packet Loss resulted in 0 percent as network transfer is within the same network . By considering all the metrics and providing evidence that numbers obtained in this thesis are statistically significant and not by

fluctuations(ANOVA and post-hoc analysis), we can better recommend Kubernetes over Docker Swarm for Web based Real Time Communication.

However, not all applications need the complex deployment, scheduling, and scaling services (or the overhead) that Kubernetes offers. But to meet the increasing demand for seamless Real time communications, and to suffice user requirements, the overhead offered by it is acceptable.

Keywords: Docker Orchestration, Kubernetes, Docker Swarm, WebRTC, Kurento, Containers, Real-Time communication, Prometheus.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisor Dr. Dragos Ilie for his encouragement and support throughout the research without whom this work would not have been possible.

I would like to thank Prof. Kurt Tutschku for conducting and managing this course efficiently.

I would also like to thank Kelvin Gakuo for the constant support during the course work through PyforKurento python module.

I am very grateful to my parents and friends especially Tarun Sai P, Vivek for their constant support and encouragement.

CONTENTS

- ABSTRACT III
- ACKNOWLEDGEMENTS V
- LIST OF TABLES..... VIII
- LIST OF FIGURES..... IX
- 1 INTRODUCTION 1**
 - 1.1 REAL TIME COMMUNICATIONS..... 1
 - 1.2 CLOUD COMPUTING..... 2
 - 1.2.1 *Cloud Service Models*..... 3
 - 1.2.2 *Virtualization*..... 4
 - 1.3 DOCKER CONTAINERS 4
 - 1.4 CONTAINER AS A SERVICE(CAAS)..... 6
 - 1.5 CONTAINER SCALABILITY 7
 - 1.6 MOTIVATION..... 8
 - 1.6.1 *Problem Statement*..... 8
 - 1.6.2 *Aims and Objectives* 8
 - 1.6.3 *Research Questions* 8
 - 1.6.4 *Defining the scope of thesis*..... 9
 - 1.7 OUTLINE 9
- 2 RELATED WORK..... 10**
- 3 CONCEPTUAL BACKGROUND 12**
 - 3.1 CONTAINER ORCHESTRATION ENGINE(COE) 12
 - 3.1.1 *Features of COE*..... 12
 - 3.1.2 *Container Orchestration Tools*..... 12
 - 3.1.3 *Cloud specific Container Services*..... 13
 - 3.2 KUBERNETES 15
 - 3.2.1 *Architecture*..... 16
 - 3.2.2 *Services and Networking* 17
 - 3.2.3 *Scaling*..... 19
 - 3.3 DOCKER SWARM 21
 - 3.3.1 *Features*..... 22
 - 3.3.2 *Architecture* 23
 - 3.3.3 *Services*..... 25
 - 3.3.4 *Networking* 25
 - 3.3.5 *Scaling*..... 26
 - 3.3.6 *Advantages* 27
 - 3.4 WEBRTC 27
 - 3.4.1 *Architecture* 28
 - 3.4.2 *Real-Time Network Transport* 29
 - 3.4.3 *API*..... 30
 - 3.4.4 *Signaling*..... 32
 - 3.5 KURENTO MEDIA SERVER..... 33
 - 3.5.1 *Architecture* 34
- 4 METHODOLOGY 36**
 - 4.1 KURENTO VIDEO CONFERENCING APPLICATION 36
 - 4.1.1 *Multiparty Architectures* 36
 - 4.1.2 *Initial Application Development and Risk State* 38
 - 4.1.3 *Alternative Chosen* 38
 - 4.2 PERFORMANCE MONITORING TOOLS 39
 - 4.2.1 *Prometheus* 39

4.3	EXPERIMENTAL SETUP	40
4.3.1	<i>Software Requirements</i>	41
4.3.2	<i>System Requirements</i>	41
4.4	IMPLEMENTATION	41
4.4.1	<i>Kubernetes</i>	41
4.4.2	<i>Docker Swarm</i>	42
4.5	EXPERIMENTAL SCENARIOS	43
4.5.1	<i>One-to-One Call Application</i>	43
4.5.2	<i>One-to-Many Call Application</i>	43
5	RESULTS AND ANALYSIS	44
5.1	CPU UTILIZATION	44
5.2	MEMORY UTILIZATION	48
5.3	NETWORK TRAFFIC	52
5.4	PACKET LOSS	56
5.5	ANALYSIS USING ANOVA	59
6	DISCUSSION	62
6.1	ATTAINING AIM OF THESIS	62
6.2	DISCUSSING RESEARCH QUESTIONS	62
7	CONCLUSION AND FUTURE WORK	65
7.1	CONCLUSION	65
7.2	FUTURE WORK	65
8	REFERENCES	66
9	APPENDIX	69
9.1	MONITORING WITH PROMETHEUS	69
9.2	ANOVA POST-HOC(STANDARD ERROR DIFFERENCES) TEST RESULTS	70

LIST OF TABLES

5.1	Average CPU Utilization per minute of Kurento Server for One-to-One call application.	45
5.2	Average CPU Utilization per minute of Kurento Server for One-to-Many call application.	46
5.3	Average Memory Utilization per minute of Kurento Server for One-to-One call application.	49
5.4	Average Memory Utilization per minute of Kurento Server for One-to-Many call application.	50
5.5	Average Network Traffic per minute of Kurento Server for One-to- One call application.	53
5.6	Average Network Traffic per second of Kurento Server for One-to-Many call application.	54
5.7	Average Packet Loss per minute of Kurento Server for One-to-One call application.	57
5.8	Average Packet Loss per minute of Kurento Server for One-to-Many call application.	57
5.9	One Way ANOVA test results for One-to-One call.	60
5.10	One Way ANOVA test results for One-to-One call.	61

LIST OF FIGURES

1.1	Server-Client Communication	1
1.2	Half Duplex RTC	1
1.3	Full Duplex RTC	2
1.4	Cloud Services	3
1.5	Docker Architecture	5
1.6	Container as a Service Representation.	6
3.1	Comparison of Containers tools with size requirement to determine the appropriate one.	15
3.2	Kubernetes Architecture	17
3.3	Kubernetes Ingress Network.	18
3.4	Cluster Autoscaler Process.	19
3.5	Horizontal Pod Autoscaler.	20
3.6	Vertical Pod Autoscaler.	21
3.7	Docker Swarm Features.	23
3.8	Docker Swarm Manager Worker Node Architecture.	23
3.9	Docker Swarm Node internals.	24
3.10	Docker Bridge and Overlay Network.	26
3.11	WebRTC Architectural Model.	28
3.12	WebRTC One-to-One call flow.	29
3.13	WebRTC Network stack.	30
3.14	WebRTC Signaling.	32
3.15	Kurento Architecture.	35
4.1	One-to-One video call signaling protocol.	37
4.2	One-to-Many video call signaling protocol.	38
4.3	Prometheus Architecture	40
4.4	Kubernetes node	42
4.5	Docker Swarm node	42
5.1	Average CPU Utilization on Kurento Server for One-to-One call(2 users)	46
5.2	Average CPU Utilization on Kurento Server for One-to-One call(14 users)	47
5.3	Average CPU Utilization on Kurento Server for One-to-Many call(2 users)	47
5.4	Average CPU Utilization on Kurento Server for One-to-Many call(8 users)	48
5.5	Average Memory Utilization on Kurento Server for One-to-One call(2 users)	50
5.6	Average Memory Utilization on Kurento Server for One-to-One call(14 users)	51
5.7	Average Memory Utilization on Kurento Server for One-to-Many call(2 users)	51
5.8	Average Memory Utilization on Kurento Server for One-to-Many call(8 users)	52
5.9	Average Network Traffic on Kurento Server for One-to-One call(2 users)	54
5.10	Average Network Traffic on Kurento Server for One-to-One call(14 users)	55
5.11	Average Network Traffic on Kurento Server for One-to-Many call(2 users)	55
5.12	Average Network Traffic on Kurento Server for One-to-Many call(8 users)	56
5.13	Average Packet Loss on Kurento Server for One-to-One call(2 users)	58
5.14	Average Packet Loss on Kurento Server for One-to-One call(14 users)	58
5.15	Average Packet Loss on Kurento Server for One-to-Many call(2 users)	58
5.16	Average Packet Loss on Kurento Server for One-to-Many call(8 users)	59
9.1	Prometheus YAML file of Kubernetes.	69

1 INTRODUCTION

1.1 Real Time Communications

The increased development and use of networked real-time applications in distributed multimedia, remote laboratory and distributed virtual reality has generated a large amount of interest in the development of real-time communication supported applications. Real time communication is the exchange of information over any type of telecommunications service instantly or with negligible latency or transmission delays from the sender to the receiver[17]. Real time communication usually has a direct path and transmits without being stored between the source and destination. Some RTC include voice over landlines and mobile phones, instant messaging, VoIP, video and teleconferencing.

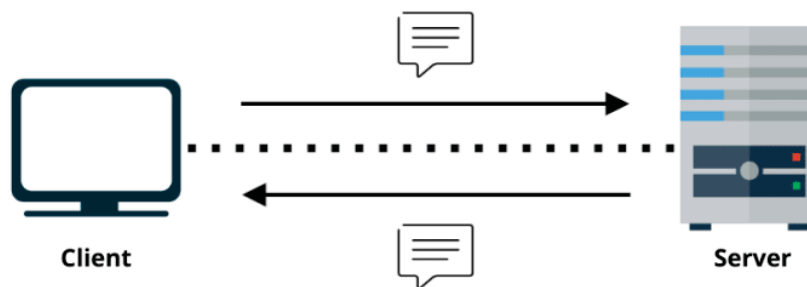


Figure 1.1 Server-Client Communication

Real time communications can take place in two modes:

- **Half-duplex RTC.** Data transmission can happen in both directions on a single carrier or circuit but not at the same time. Either the sender or receiver can send but not receive at the same time.

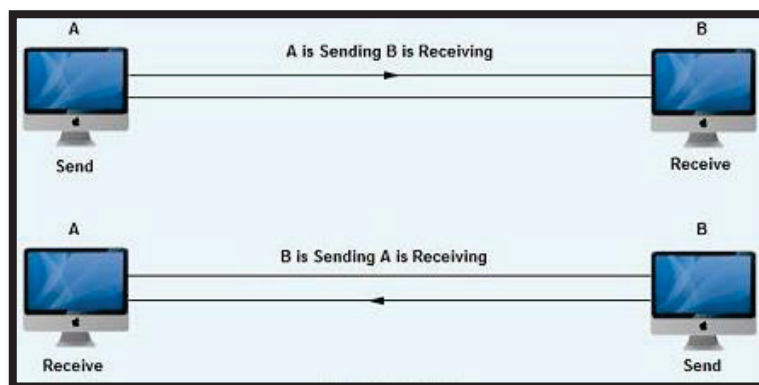


Figure 1.2 Half Duplex RTC

- **Full-duplex RTC.** Data transmission can occur in both directions simultaneously in two parallel communication pathways.

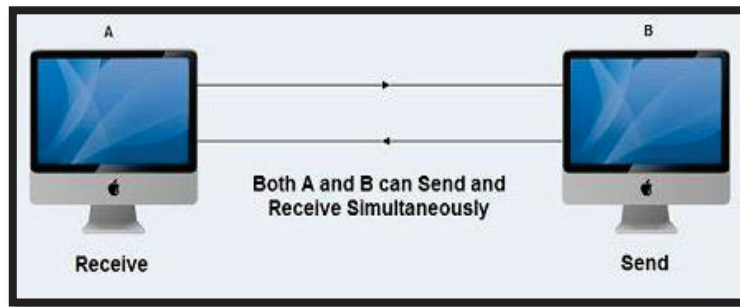


Figure 1.3 Full Duplex RTC

The evolution of internet, mobiles and smart devices in the late 20th century urged for further development in real-time communications. Today's users expect to connect in real time from any location and device, making RTC a critical element of unified communications. Multimedia has been the part and parcel of life. Being in contact with each other such as collaboration, instant messaging, surveillance, IoT, entertainment, connected and self-driving automobiles, navigation, live updates on events around the world created an immediacy in various fields such as customer service, healthcare, security, education and more. Even the Covid-19 pandemic around the world in 2019-2020 has changed the perspective of real-time communication. Individuals switched to the audio and video means of communication rapidly. Organisations are engaged virtually from almost anywhere by video conferencing, file sharing for maximising the productivity and flexibility. This has created a huge impact and need on Telecommunication industry to scale the resources to provide uninterrupted services.

This increase demand in Real-time communication has to offer simple and standard solutions which are feasible, interoperable, scalable and cost-effective. On the other hand, as we have seen, people are encouraged to communicate through web services. The traditional methods of web communications were modified to adapt cost-effective solutions. One such solution for web based real-time communication is WebRTC, open-sourced by Google in 2011. According to IEEE Standards Communications Magazine, WebRTC is a joint effort of Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C) to standardize real time communications across the internet.

WebRTC helps for the Real time communication capabilities between Server and a client using simple JavaScript RestAPIs. This eliminates the security and performance concerns which come with tools such as Adobe Flash player and various other browser plugins as WebRTC enables us to access multimedia with plug-in less approach. The technology is available on all modern browsers and it supports audio, video and generic data transmission between peers offering voice and video communication solutions[18].

There are many WebRTC frameworks such as YouPHPTube, EasyRTC, SimpleWebRTC etc. but Kurento Media Server is chosen for following reasons. Kurento Media Server is an open-source WebRTC server which provides us with simple set of APIs'. It simplifies the process of developing Video applications for mobile, IoT, Web devices. Kurento Media server also comes with features like group communications, transcoding, recording, mixing, broadcasting and routing of audio-visual flows. As a differential feature, Kurento Media Server also provides advanced media processing capabilities involving computer vision, video indexing, augmented reality and speech analysis. Kurento Media plain has been created independently basing on the rich multimedia features provided by the G-Streamer project. The main advantage of Kurento is its versatility by introducing the concept of Media workflow that allows to define in code how and where the media flows.

1.2 Cloud Computing

In 1997, Chellappa first-ever coined the term “Cloud Computing” (CC) in his address to the INFORMS Annual Meeting. His idea was more concentrated toward usage of cloud services in terms of economical

rationalization among Small and Medium-sized Enterprises (SMEs) or upcoming business models which earlier was never thought of[4]. Cloud Computing technology uses the internet and central remote servers to maintain data and applications[6]. Cloud computing has come a long way with various technological advancements in the past two decades. It has been accelerated with the evolution of various virtualization technologies. Clouds are capable with the scalability and flexibility metrics. Various companies such as Amazon, Microsoft provide cloud Infrastructure as a Service for their customers to run their code and deploy their applications. Applications of CC are now widespread with mailing services to storage, document processing, hosting services, image processing, and video streaming. However, problems associated with current CC are multifold such as: reliability, availability of services and data, security, complexity, limited customization, and cost of service[4].

1.2.1 Cloud Service Models

The three main types of cloud computing service models include Infrastructure as a Service, Platform as a Service and Software as a Service. Each type of cloud computing provides different levels of control, flexibility, and management so that one can select the right set of services for their needs[5].

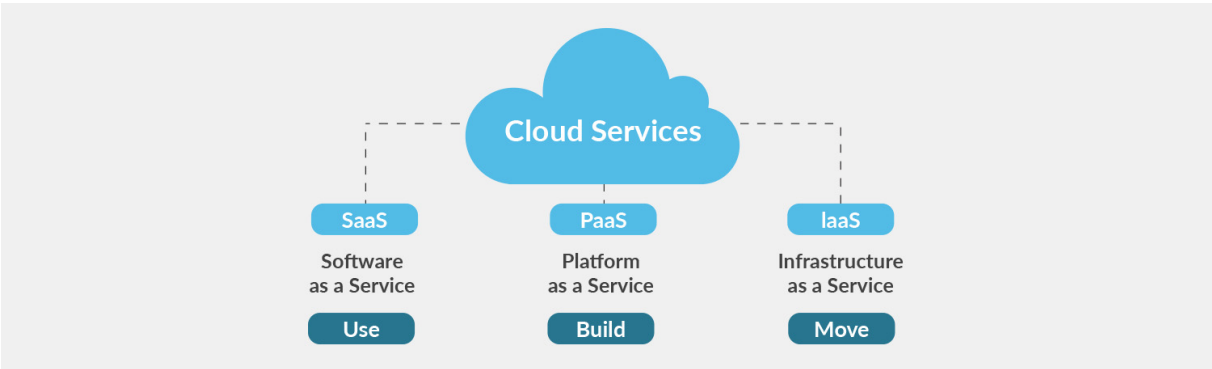


Figure 1.4 Cloud Services

Infrastructure as a Service (IaaS)

IaaS contains the basic building blocks for cloud IT. It typically provides access to networking features, computers (virtual or on dedicated hardware), and data storage space. IaaS gives us the highest level of flexibility and management control over the IT resources. It is most similar to the existing IT resources with which many IT departments and developers are familiar.

Platform as a Service (PaaS)

PaaS removes the need for people to manage underlying infrastructure (usually hardware and operating systems) and allows to focus on the deployment and management of the applications. This helps users to be more efficient as they do not need to worry about resource procurement, capacity planning, software maintenance, patching, or any of the other undifferentiated heavy lifting involved in running the application.

Software as a Service (SaaS)

SaaS provides with a complete product that is run and managed by the service provider. In most cases, people referring to SaaS are referring to end-user applications (such as web-based email). With a SaaS offering, people don't have to think about how the service is maintained or how the underlying infrastructure is managed. They only need to think about how that particular software is used.

1.2.2 Virtualization

Cloud computing, and particularly the use of public clouds, brings advantages on the technical, environmental and business sides, allowing multiple under-utilized systems to be consolidated within fewer physical servers hosting them. A cloud provider can manage physical resources in a very efficient way by scaling on the several hundreds and thousands of customers with dynamically changing workload requirements, by re-optimizing the infrastructure in a completely automated (or semi-automated) fashion whenever needed, providing high levels of availability and reliability. One of the most important technologies that enabled this paradigm shift in computing is virtualization, and particularly machine virtualization[7].

Virtualization is the key component of cloud computing. Virtualization technology offers applications an abstract view through interfaces of the underlying hardware platform and resources[7]. Machine virtualization (also referred to as processor virtualization) allows a single physical machine to emulate the behavior of multiple machines, with the possibility to host multiple and heterogeneous operating systems (called guest operating systems or guest OSs) on the same hardware. A virtual machine monitor (VMM), or hypervisor, is the software infrastructure running on (and having full control of) the physical host and which can run such emulation.

Containers offer another approach to implement virtualization at the operating system level. A container engine can run applications whose binaries have been encapsulated into a container image together with the required library dependencies and configuration files. Different kernel process groups and namespaces are used in the single shared kernel to isolate different containers running on the same physical processing node[12]. Due to their intrinsic portability, containers have been used to support the development, testing, deployment and management of complex cloud-based and non-cloud-based applications. They provide an appropriate substrate to decompose large monolithic applications into independent micro-services [12], each packed into different container images. These micro-services can then be individually deployed and configured accordingly, to cooperatively provide the application's functionalities. The micro-service architecture also provides a convenient way to address non-functional requirements such as overall performance, high availability and scalability[12].

Docker is the popular container engine among others such as CRI-O, Railcar, rkt(rocket), Linux containers(lxc/lxd). Docker containers offer better capabilities than Virtual machines. These have the large scalability, safe to develop and utilize as compared to previous virtual machine technique. Docker technology is light weight and fast to launch, run, stop, restart containers with superior execution, little isolation and more concord. However, VMs have an entire operating system including its own memory installed and the associated overhead for drivers of virtual device[8].

1.3 Docker Containers

Docker containers right now are playing a very important role in the field of cloud computing. Docker is a container engine written in Go which renders the lightweight virtualization at system level through extending a common container format approach on Linux called Linux Containers (LXC)[8]. It uses namespace technology to provide isolation. Each aspect of container runs in a separate namespace and its access is limited to that namespace.

Docker is a uniting block for automating distributed systems: database clusters, large scale web deployments, continuous organization systems, creation of lightweight, private PAAS environments, service-oriented architectures, packaging automation and applications development, testing automation and continuous deployment/integration, scaling and deploying web apps, backend and databases services. They deliver multiple applications hassle free and have them run the same way on all our environments - desktop or cloud-native.

Dockers come with various advantages such as:

- High portability and ability to run anywhere
- Cost-effective
- Scalable
- Share same operating systems

These advantages of Docker containers have made them a good viable option instead of Virtual machine which share the Hypervisor space. Since VMs have a macro view and can operate on most modern operating systems as well, it is of utmost importance for a cloud architect to choose between these two technologies and decide the deployment procedure based on each application.

Docker Architecture

Docker uses a client-server architecture. The Docker client talks to the Docker daemon responsible for building, running, and distributing the Docker containers. The Docker client and daemon can run on the same system or on a remote system. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.

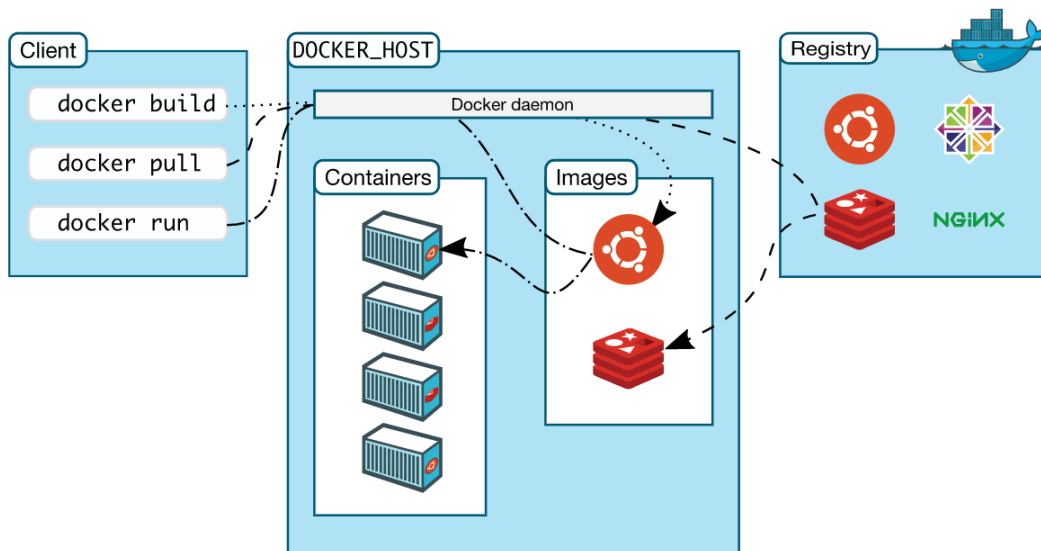


Figure 1.5 Docker Architecture

Docker Daemon

The Docker daemon listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

Docker Client

The Docker client is the primary way that many Docker users interact with Docker. Docker uses the Docker API to communicate with docker daemon through simple commands, which carries them out. The Docker client can communicate with more than one daemon.

Another Docker client is Docker Compose[14], that enables us to work with applications consisting of a set of containers. Docker Compose is developed to help, define and share multi-container application stack in a file. With Compose, we can create a YAML file to define the services with a single command.

Docker registries

A Docker registry stores Docker images. Docker Hub is a public registry and Docker is configured to look for images on Docker Hub by default. We can even run our own private registry where modified images can be stored.

Docker Images

An image is a read-only template with instructions for creating a Docker container. We can build our own image through a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When we change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies[14].

A container is a runnable instance of an image. We can create, start, stop, move, or delete a container using the Docker API or CLI. We can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

1.4 Container as a Service(CaaS)

Container as a Service (CaaS) is a business model whereby cloud computing service providers offer container-based virtualization services as a scalable online service. This allows users to use container services without having the necessary infrastructure. User interface and orchestration are treated as main Differentiators[15].

Most CaaS solutions do not include development tools unlike PaaS. They just provide a complete stack of tools for deploying containers. Those tools include the Docker runtime, a container image registry, an orchestrator and infrastructure to host it all. A CaaS provides value by offering an easy on-ramp for setting up and running a Docker environment. In contrast, a PaaS caters to a different set of use cases, which involve application development and deployment[16]. PaaS and CaaS can blend in some scenarios where we need to both develop and deploy the applications to containers.

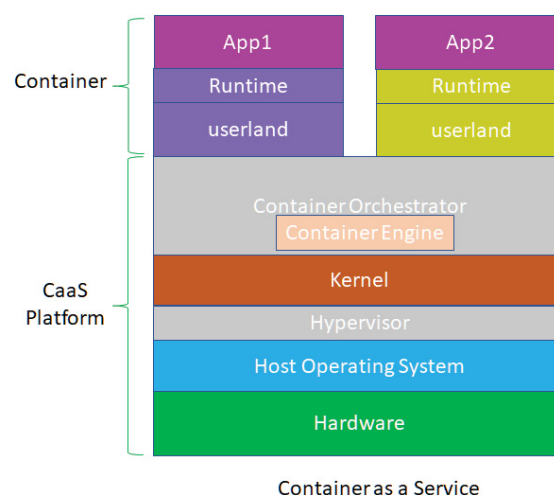


Figure 1.6 Container as a Service Representation

Most of the advantages offered by CaaS are similar to stand-alone containers. However, we have few benefits specific to cloud services:

- **Flexibility:** Applications that were designed for on-premises use do not behave as expected when installed on a cloud-based virtual machine. So, CaaS makes it easier to run applications in cloud.
- **Agility:** Agility is the ability to roll out a new production workload as quickly as possible.

Multi-cloud orchestration[11] is in use for different reasons, as the benefits can include infrastructure cost optimization, flexibility and portability (including reducing vendor lock-in), and scalability (such as dynamically scaling out a cloud from an on-premises environment when necessary). Multi-cloud container orchestration refers to the use of an orchestration tool to operate containers across multi-cloud infrastructure environments instead of running containers in a single cloud environment. Multi-cloud environments and containers go hand-in-hand because of the latter's portable, "run anywhere" nature.

1.5 Container Scalability

Software that was considered scalable yesterday, may now be perceived as inflexible and overly entangled compared to the suites of so-called microservices that are today widely used. Microservices are loosely coupled, independently deployed, cloud-native services[21]. They are a realization of the service-oriented architectural style of building software composed of small services that has separate business functionality, running as a separate process, and communicate with each other through lightweight mechanisms. As they are loosely coupled and the failure of one microservice does not affect other microservices of the system, they are adopted over monolithic applications[22]. Microservice architectures facilitate scalability since new instances of the same microservice can be launched to split the workload locally, without scaling the overall service[21].

Container scalability is the trait where a container application can handle increased loads of work. The problem started to raise with the increase in demand for containerizing the large and complex applications running numerous microservices. The main problems by using the container engine itself are clustering and scalability. The applications are entitled to run on a single host in above case and if the server fails, the whole application comes down due to lack of clustering of servers. On the other hand, if there are more traffic requests hitting the application at a given time, server needs to scale the resources to accommodate the incoming traffic. This can be achieved by reconfiguring the existing architecture of a single machine to increase available resources or by provisioning additional containers within a cluster of distributed machines. Containers offer better scalability and maintainability but scaling up and down with docker engine is not feasible. So, to manage large set of applications and address these problems, container orchestration came into picture.

Container orchestration is the automatic process of managing or scheduling the work of individual containers for applications based on microservices within multiple clusters[10]. Container orchestration works with tools like Kubernetes, Docker Swarm, Apache Mesos and Cattle[53].

Kubernetes is an open-source platform used for automatic deployment, scaling and management of containerized applications. It comes with container-centric management environment. It provides Platform as a service (PaaS) with the flexibility of Infrastructure as a service thus enables its users a chance of portability across various infrastructure providers. It provides tools for Application-specific workflows which can be helpful for faster development of applications. Kubernetes also provides with its control plane facility which provides APIs for both user and developer to customize their own controls.

Docker Swarm is also an open-source container orchestration platform. Applications which use Docker containers can be deployed in Swarm equally well. Swarm utilizes the same command line from Docker. Docker Swarm, when compared to Kubernetes, can deploy container much easier and faster which allows for faster reaction times[19]. It offers high availability. Swarm mode consists of DNS element that can be utilized for distributing incoming requests to service name.

1.6 Motivation

1.6.1 Problem Statement

Currently almost every social platform and small-scale applications look towards cloud to deploy their services successfully and provide maximum satisfaction to their end-user. Thus, virtualizing their services becomes utmost important to deploy and develop their applications. This alone emphasizes the importance of Docker containers in the development world. Since Multimedia plays a huge role in day to day lives and most people crave for faster and efficient responses, it is essential to develop our applications with better Real time communication capabilities. Thus, we are determining which container orchestration tool serves best for Real time communication applications. So, a multimedia(video conferencing) application is developed and is deployed using Kurento media server and measure the performance of the server when the application is deployed. We have chosen Kubernetes and Docker Swarm as container platforms for this thesis. The Servers and Clients are virtualized and metrics such as CPU Utilization, Network Traffic, Packet Loss, Memory Utilization will be measured along with Container overhead. These metrics provide the performance overhead in different scenarios for each orchestration technology. This will be helpful to analyze and understand the effect of Kurento server on these technologies. Thus, the results are expected to determine which orchestration technology serves best for RTC applications.

1.6.2 Aims and Objectives

The aim of this work is to investigate the performance of WebRTC when used with different container technologies (Kubernetes and Docker Swarm). We are interested in estimating the performance overhead using the containers mentioned above. The objectives of this project are:

- To implement WebRTC based Kurento server in a container environment.
- To extract performance metrics such as Network Traffic, CPU and Memory Utilization using containers.
- To compare WebRTC based Kurento server in different container technologies.

1.6.3 Research Questions

The following questions will be answered in our thesis after performing multiple experiments, analysing with relevant metrics and quantifying the effectiveness of the solution.

1. How does the performance of Kubernetes and Docker Swarm container technologies get affected when application is scaled?
2. Is there any significant performance overhead between Kubernetes and Docker Swarm in terms of CPU Utilization, Memory Utilization and Network Traffic?
3. Which of the above-mentioned container orchestration technologies is best suited for Web based Real-Time Communication based on previous analysis?

1.6.4 Defining the scope of thesis

The scope of this thesis is limited to analyzing the scalability of Kurento clients using Docker based Orchestration tools Kubernetes and Docker Swarm, testing the performance of various metrics in non-cloud-based environment. This paper helps to choose better orchestration tool using docker daemon for Real time communication using WebRTC based Kurento framework in terms of CPU and Memory Utilization along with Network Traffic and overhead.

The thesis does not consider High availability of resources or other performance parameters. Security is not considered in this work in order to simplify the building of the system.

1.7 Outline

- Chapter 1 i.e., this chapter consists of the basic overview of the thesis along with the background knowledge necessary including Real time communications, Docker, Kubernetes, Docker Swarm.
It also includes the motivation for the thesis, the problem statement, research questions, Aims and Objectives.
- Chapter 2 deals with the related work which tells us about the existing works done related to Docker, Orchestration tools, WebRTC and Kurento as a part of the Literature study.
- Chapter 3 deals with the conceptual background in the thesis to setup and perform the experiments necessary.
- Chapter 4 deals with the methodology adopted in the thesis to setup the virtualised and containerised environment to perform the tests necessary for the same.
- Chapter 5 shows all the results recorded during the experiments performed and statistical analysis of the obtained results.
- Chapter 6 discusses the answers to the research questions obtained during the thesis, other possibilities of research.
- Chapter 7 gives the conclusion i.e., summing up the thesis work and presenting the perspectives and possible future research that can be done to extend the existing thesis work.
- Chapter 8 provides the references which were helpful during the thesis.
- Chapter 9 ends the document with Appendix section.

2 RELATED WORK

In paper [1] authors focus on analysis of video quality and end-to-end latency in WebRTC. They describe WebRTC as a set of emerging technologies that extends the web browsing model to exchange real-time media with other browsers. In spite of the fact that WebRTC is still in under development, it is gaining the attention of practitioners quickly. The mechanisms to provide quality assurance for WebRTC are key to release these kind of applications to production environments. Here in this article authors present the Kurento Testing Framework (KTF), a piece of software aimed to simplify the evaluation activities for WebRTC applications and services. It provides advanced features to carry out complete assessment of WebRTC applications in terms of functionality and quality-of-experience.

Emiliano Casalicchio et.al Vanessa Perciballi et.al in paper [2], focuses on Measuring Docker Performance. They say internet of service industry is adopting the container technology both for internal usage and as commercial offering. The use of container as base technology for large-scale systems opens many challenges in the area of resource management at run-time, for example: autoscaling, optimal deployment and monitoring. Specifically, monitoring of container-based systems is at the ground of any resource management solution, and it is the focus of this work. This paper explores the tools available to measure the performance of Docker from the perspective of the host operating system and of the virtualization environment, and it provides a characterization of the CPU and disk I/O overhead introduced by containers.

Cristian Constantin et.al Spoiala, Corneliu Octavian Turcu, et.al in paper [3] describe the Current developments in real-time technologies that enable multiple companies to focus on a standard, in order to bring real-time technologies to the web. The efforts lead to the API standard WebRTC that supports voice and video chat and P2P file sharing, without the need to install external plugins on browsers. As lots of services will switch to WebRTC they found it is useful to provide a comparison of a WebRTC server on Docker containers and virtual machines. As WebRTC server, they used Kurento Media Server, a powerful open-source server with many advanced features. This paper is testing what kind of virtualization is more suitable for a multimedia application based on WebRTC. They tested Docker containers and KVM with a multimedia-based test.

In this thesis, we concentrated on One-to-One and One-to-Many video call topology whereas Bart Jansen et.al performed a thorough evaluation of WebRTC focusing on the Google Congestion Control(GCC) algorithm, both in emulated synthetic network conditions as well as in real wired and wireless networks. In the synthetic network conditions, they introduced latency, Packet Loss and minimised bandwidth to study the impact. According to them WebRTC streams have a slightly higher priority than TCP flows when competing with cross traffic. They compared the performance of a mesh(N-N call)[51] and Selective Forwarding Unit (SFU) based topologies for group video calls using WebRTC. They mentioned adding an SFU can significantly improve the performance of multi-party video call. Finally, WebRTC utilizes the Google Congestion Control (GCC) algorithm to provide congestion control for real-time communications over UDP. It is also concluded that WebRTC is sensitive to variations in RTT and packet losses.

Eddy Truyen et.al Dimitri Van Landuyt et.al in paper [49] explain Container orchestration frameworks provide support for management of complex distributed applications. Different frameworks have emerged only recently, and they have been in constant evolution as new features are being introduced. This reality makes it difficult for practitioners and researchers to maintain a clear view of the technology space. They presented a descriptive feature comparison study of the three most prominent orchestration frameworks: Docker Swarm, Kubernetes, and Mesos, which can be combined with Marathon, Aurora or DC/OS. Their study aims at (i) identifying the common and unique features of all frameworks,(ii) comparing these frameworks qualitatively and quantitatively with respect to genericity in terms of supported features, and (iii) investigating the maturity and stability of the frameworks as well as the pioneering nature of each framework by studying the historical evolution of

the frameworks on GitHub. They came up with Results, where they have identified 124 common features and 54 unique features that they divided into a taxonomy of 9 functional aspects and 27 functional sub-aspects, Kubernetes supports the highest number of accumulated common and unique features for all 9 functional aspects; however, no evidence has been found for significant differences in genericity with Docker Swarm and DC/OS, very little feature deprecations have been found and 15 out of 27 sub-aspects have been identified as mature and stable. These are pioneered in descending order by Kubernetes, Mesos and Marathon.

Vilhelm Friðriksson evaluated the Overhead introduced by Docker in microservice systems[53]. He tried to close the gap by performing evaluations on a small microservice system running on virtual machines, both with and without container technology. He also tried to explain the nature of the overhead incurred for containerized microservices. The system performed worse on every load test for the Docker setup when compared to the native setup. The overhead was at its minimum when the system was under the least load and may be considered somewhat acceptable. Comparison of the service time median values, provided by Taurus, showed that at its minimum the overhead of using Docker was 8.7% and at its maximum it was 62.5% for the multiple server system. The same comparison for the single server setup showed the overhead to be 24.92% at its minimum and increasing up to 59.09%. These calculations exclude the extreme cases when the Docker system had reached its maximum capacity, but the native system had not. In those cases, the overhead was 479247.62% for the single server setup and 177282.14% for the multiple servers setup.

To evaluate the performance of the cloud based container orchestration tools[12] Yao Pan et.al adopted two step strategy. In the first step they used a test suite to measure performance of a single container running on a single worker node. This helped them to analyze overheads that are introduced at each worker node by each container orchestration tools. Then they used an application that is implemented by multiple containers running over a cluster of cloud-based worker nodes. This helped them to analyze the overheads when containers were communicating with each other. In both the cases, they have experimented by manually configuring the system using the underlying container technology. By comparing the performance of this baseline setting against the performance of orchestration tools, they estimated performance overhead associated with the tool. The results show that performance of Kubernetes is slightly worse than Docker Swarm, but Docker Swarm is not as flexible as Kubernetes in complex situations.

3 CONCEPTUAL BACKGROUND

3.1 Container Orchestration Engine (COE)

Container orchestration engine automates the deployment, management, scaling, and networking of containers. Enterprises that need to deploy and manage hundreds or thousands of containers and hosts can benefit from container orchestration[9].

Container orchestration can be used in any environment where we use containers. It can help us to deploy the same application across different environments without needing to redesign it and microservices in containers make it easier to orchestrate services, including storage, networking, and security. Containers give our microservice-based apps an ideal application deployment unit and self-contained execution environment. They make it possible to run multiple parts of an app independently in microservices, on the same hardware, with much greater control over individual pieces and life cycles.

Managing the lifecycle of containers with orchestration also supports DevOps teams who integrate it into CI/CD workflows. Along with application programming interfaces (APIs) and DevOps teams, containerized microservices are the foundation for cloud-native applications.

Most of the Container Orchestration Engines perform two primary functions, clustering and scheduling. The way clustering forms in most of the tools are by having a Master node where the orchestration engine is installed and configured and then we join with the Worker node. The combination of master node and worker nodes is called the cluster of nodes. Master node acts as cluster manager and manages the worker nodes inside that cluster. Based on requirements provided to COE through files, it schedules the tasks and assign them to suitable nodes in cluster. This is called scheduling.

3.1.1 Features of COE

- Provisioning and deployment
- Configuration and scheduling
- Resource allocation
- Container availability
- Scaling or removing containers based on balancing workloads across the infrastructure.
- Load balancing and traffic routing
- Monitoring container health (Fault tolerance)
- Configuring applications based on the container in which they will run.
- Keeping interactions between containers secure.

3.1.2 Container orchestration tools

Container orchestration tools provide a framework for managing containers and microservices architecture at scale. There are many container orchestration tools that can be used for container lifecycle management. Some popular options are Kubernetes, Docker Swarm, and Apache Mesos. Configurations files tell the container orchestration tool how to network between containers and where to store logs. The orchestration tool also schedules deployment of containers into clusters and determines the best host for the container. After a host is decided, the orchestration tool manages the lifecycle of the container based on predetermined specifications. Container orchestration tools work in any environment that runs containers.

Benefits of Container orchestration tools:

- Increased portability: Scale applications with a single command and only scale specific functions without affecting the entire application.
- Simple and fast deployment: Quickly create new containerized applications to address growing traffic.
- Enhanced productivity: Simplified installation process and decreased dependency errors.
- Improved security: Share specific resources without risking internal or external security. Application isolation improves web application security by separating each application's process into different containers.

3.1.3 Cloud Specific Container Services

The most popular cloud providers in the market are Google, Amazon and Microsoft. They offer services to manage containers at a large scale.

Google Kubernetes Engine(GKE)

Google Kubernetes Engine (GKE) provides a managed environment for deploying, managing, and scaling the containerized applications using Google infrastructure. The GKE environment consists of multiple machines (specifically, Compute Engine instances) grouped together to form a cluster[37].

GKE clusters work with the help of Kubernetes open-source cluster management system. Kubernetes commands and resources are used to deploy and manage the applications, perform administration tasks, set policies, and monitor the health of the deployed workloads.

By using a GKE cluster to run the application, we also gain the benefit of advanced cluster management features that Google Cloud provides. These include load-balancing, Node pools, Automatic scaling, Automatic upgrades, Node auto-repair and Logging and monitoring. GKE cluster control planes are automatically upgraded to run new versions of Kubernetes or can be manually upgraded by initiating a control plane upgrade from the open source of Kubernetes project. New features in Kubernetes are generally named Alpha, Beta or Stable. Kubernetes features Beta or Stable are included with GKE whereas Kubernetes Alpha features are available in special GKE Alpha clusters.

GKE containerized applications are packaged into platform independent, isolated user-space instances, for example by using Docker which is supported by GKE. In GKE and Kubernetes, these containers, whether for applications or batch jobs, are collectively called workloads. Before we deploy a workload on a GKE cluster, we must first package the workload into a container.

Google Cloud provides continuous integration and continuous delivery tools to help us build and serve application containers. We can use Cloud Build to build container images (such as Docker) from a variety of source code repositories, and Artifact Registry or Container Registry to store and pull the container images.

GKE clusters have two modes of operation. They are Autopilot and Standard. Autopilot mode of operation manages the entire cluster and node infrastructure. These clusters are pre-configured with an optimized cluster configuration that is ready for production workloads. Standard mode of operation provides node configuration flexibility and full control over managing the clusters and node infrastructure.

ECS Container Services

Amazon Elastic Container Service (Amazon ECS) is a fully managed container orchestration service. Organizations such as Duolingo, Samsung, GE, and Cookpad use ECS to run their most sensitive and mission critical applications because of its security, reliability, and scalability[38].

ECS run containers on ECS clusters using AWS Fargate, which is a serverless compute for containers. ECS is used extensively within Amazon to power services such as Amazon SageMaker, AWS Batch, Amazon Lex, and Amazon's recommendation engine, ensuring it for security, reliability, and availability. ECS has been a foundational pillar for key Amazon services as it can integrate with other services providing a familiar experience to deploy and scale the containers. It is also able to integrate with other AWS services to bring new capabilities to ECS. For example, ECS allows applications the flexibility to use a mix of Amazon EC2 and AWS. ECS also integrates with AWS App Mesh, which is a service mesh, to bring rich observability, traffic controls and security features to the applications. ECS has grown rapidly since launch and is currently launching 5X more containers every hour than EC2 launches instances.[38]

ECS provides various benefits like Serverless option, High security, Application with Capacity Providers, highly scalable services, Reliable service and optimized for cost. Amazon ECS works by building images and storing them using Elastic Container Registry (ECR) or any other repository. These images are deployed in Amazon Elastic Container Service by selecting container images and resources needed for the application and by launching containers on supporting compute services such as AWS Fargate, AWS Regions, AWS Local Zones, ECS Anywhere and so on. Amazon ECS scales the application and manages the containers for availability.

ECS can also be helpful in Hybrid deployment, Batch processing, Machine learning and Web applications.

Azure Container Services

Microsoft Azure offers container services such as Azure Kubernetes Service (AKS), App Service, Service Fabric, Azure Red Hat OpenShift. Azure Kubernetes Service (AKS) simplifies deploying a managed Kubernetes cluster in Azure by offloading the operational overhead to Azure. As a hosted Kubernetes service, Azure handles critical tasks, like health monitoring and maintenance. Since Kubernetes masters are managed by Azure, we only manage and maintain the agent nodes[39].

AKS cluster can be created using The Azure CLI, The Azure portal, Azure PowerShell and Template-driven deployments like Azure Resource Manager templates and Terraform. When an AKS cluster is deployed, the Kubernetes master and all nodes are deployed and configured. It can also support Windows Server containers. To enhance the security and management, AKS helps to integrate with Azure Active Directory (Azure AD) to use Kubernetes role-based access control (Kubernetes RBAC) and monitor the health of the cluster and resources. Kubernetes RBAC controls access and permissions to Kubernetes resources and namespaces.

Azure Monitor for Container Health collects memory and processor performance metrics from containers, nodes, and controllers within then AKS cluster and deployed applications. This can be reviewed in both container logs and the Kubernetes master logs.

AKS nodes run on Azure virtual machines (VMs). With AKS nodes, we can connect storage to nodes and pods, upgrade cluster components, and use GPUs. AKS supports Kubernetes clusters that run multiple node pools to support mixed operating systems and Windows Server containers[39].

The number of cluster nodes or pods that run services automatically scales up or down by adjusting both the horizontal pod autoscaler or the cluster autoscaler to use only the necessary resources. AKS offers multiple Kubernetes versions. New versions can be upgraded using the Azure portal or Azure CLI. During the upgrade process, nodes are carefully cordoned and drained to minimize disruption to running applications.

AKS supports the creation of GPU-enabled node pools, Intel SGX-based confidential computing node pools (DCSv2 VMs) and storage volume. An AKS cluster can be deployed into an existing virtual network. This helps to communicate with other pods in the cluster and other nodes in the virtual network. Pods can also connect to other services in a peered virtual network and to on-premises networks over ExpressRoute or site-to-site (S2S) VPN connections. Azure provides several tools that help streamline Kubernetes, such as Azure Dev Spaces and DevOps Starter. AKS also supports Docker image format. For private storage of Docker images, we can integrate AKS with Azure Container Registry (ACR).

Based on the features of each tool, considering the needs of video conferencing application and the penetration in today’s market, in this thesis we have decided to proceed with Kubernetes and Docker Swarm for evaluation.

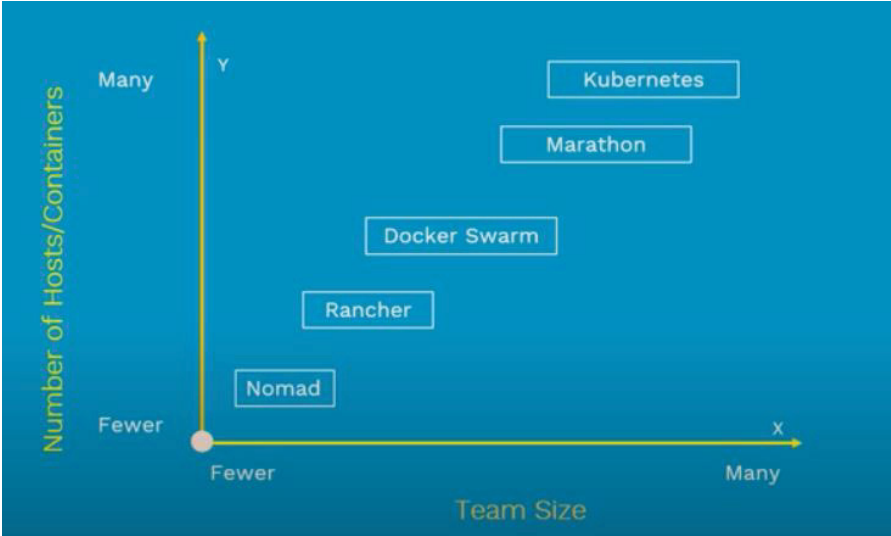


Figure 3.1 Comparison of Containers tools with size requirement to determine the appropriate one

3.2 Kubernetes

Kubernetes is a production grade container orchestration solution developed and released by Google in 2014. It is now hosted by the Cloud Native Computing Foundation(CNCF). It is a modular system that allows different components and plugins to be included in a Kubernetes cluster[12]. Kubernetes has a control plane and nodes which are a set of machines. These communicate with each other through the API server. Kubernetes is based on a master/slave architectural pattern for which a developer submits a list of applications to a master node, the platform deploys them across slave and master nodes[53]. Slave is also referred as Worker node.

Kubernetes supports several container runtimes other than Docker such as containerd, CRI-O and any Kubernetes CRI implementation. Any application running on Kubernetes is referred as workload. Workload can be a single component or several containers that work together. These are represented as Pod, a set of running containers on Kubernetes cluster. They have a defined lifecycle, managed by workload resources instead of us monitoring each one. Workload resources configure the controllers.

Kubernetes provides many in built workload resources. Deployment manages the application workload on cluster which is comprises of Pods. Pods are interchangeable and can be replaced if needed. ReplicaSet is another workload resource which replaces the ReplicationController from previous versions. Other workload resources are namely Statefulset, DaemonSet, Job and CronJob. They perform different functions enabling us to run the large and complex considerable easier.

Kubernetes provides us benefits like automatic management, monitoring and liveness probes for application containers, automatic scaling, rolling updates, and more.

3.2.1 Architecture

A cluster is a group of nodes or servers. A cluster should at least comprise of one node. The cluster's brain is called the control plane, and it runs all the tasks required for Kubernetes. Nodes in a cluster are each given a role, one node functions as the master node. It runs control plane, implements a server that acts as a gateway and controller for the cluster by exposing an API for developers and external traffic. The Kubernetes Master is the core of the Kubernetes orchestrator. It carries out scheduling and orchestrates communication between other components. The master node acts as the primary point of contact with the cluster and is responsible for most of the centralized logic that Kubernetes provides.

Kubernetes master consists of the following key components:

- **API Server** is the central component that acts as a management hub. It serves the Kubernetes API and communicates with all other components.
- **etcd** is a lightweight, key-value data store component that stores all configuration data including a registry of what is running and where it is running.
- **The Scheduler** maintains the availability, performance and capacity of the cluster.
- **A Controller Manager** monitors and watches etcd via the API server. It consists of several controllers, which run in separate processes, for instance:
- **Node Controller**, which manages, discovers, monitors all nodes in the cluster, and responding to the health condition of nodes.
- **Replication Controller**, which maintains the replicated objects in the cluster.
- **Endpoint Controller** ensures the service for each pod is always up-to-date and manages endpoints.
- **Service Account and Token Controllers**, manages service accounts and API access tokens for new namespaces.

The Kubernetes nodes, also known as worker nodes, receive requests from the Master node. The workers host pods and form the larger part of a Kubernetes cluster. The worker nodes have explicit resource capabilities, which are known by the system. These are given as a set of labels attached to a worker node to specify its version, status and particular features.

Each Kubernetes node consists of the following components:

- **Kubelet** is the node agent that runs on each user node.
- **kube-proxy** runs on each node to provide simple network proxying and load balancing.
- **Labels** is a key-value pair attached to objects such as services or nodes.
- User **pod** consists of container(s), which have been deployed or scheduled in groups.

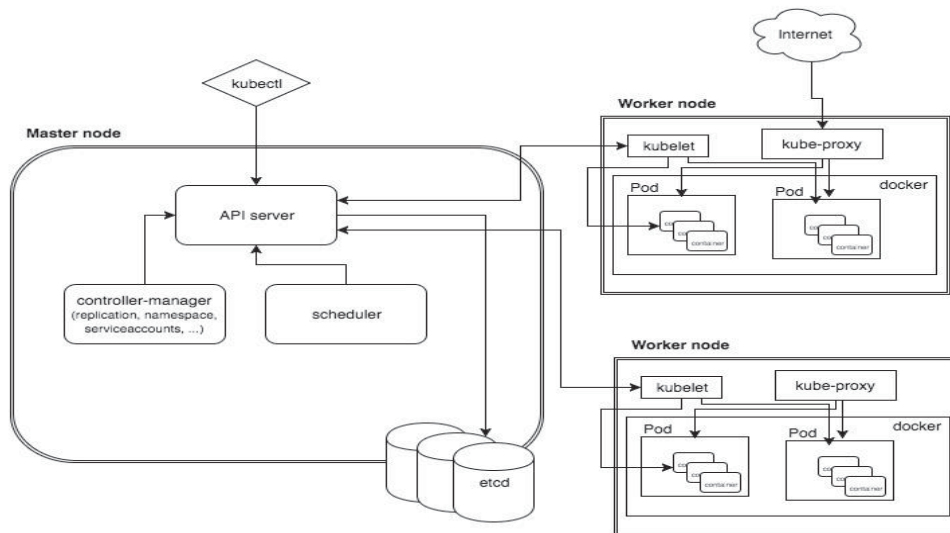


Figure 3.2 Kubernetes Architecture

Pods are the basic scheduling unit in Kubernetes. They are high-level abstractions for groups of containerized components. A pod consists of one or more containers that are guaranteed to be co-located on the host machine and can share resources. A pod is deployed according to its resource requirements and has its own specified resource limits. For two or more pods to be deployed in the same node, the sum of the minimum amounts of resources required for the pods needs to be available in the node. All pods have unique IP address, which allows applications to use ports without the risk of conflict. Within the pod, containers can reference each other directly, but a container in one pod cannot address a container in another pod without passing through a reference to a service, the service holds a reference to the target pod at the specific pod IP address. The IP addresses of pods are not constant and are reassigned on pod creation and system boot[21].

Scheduler is in charge of assigning pods to specific nodes in the cluster. The scheduler matches the operating requirements of a pod's workload to the resources that are available in the current infrastructure environment, and places pods on appropriate nodes. The scheduler is responsible for monitoring the available capacity on each node to make sure that workloads are not scheduled in excess of the available resources. The scheduler needs to know the total capacity of each node as well as the resources already allocated to existing workloads on the nodes[21].

Kubernetes supports multiple master nodes running simultaneously, but only one controller manager and scheduler can perform actions within one of the master nodes of the cluster. When the current master dies, a new master is elected.

3.2.2 Services and Networking

Service is an abstraction of exposing the application running in pods as a network service to the outside environment. They act as basic internal load balancers and ambassadors for pods. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods and can load-balance across them. Service in Kubernetes is a REST object, and we can post service definition to API server to create new instance. We have to make sure that the name of service object must be valid DNS label name. Each Service is assigned a unique IP address (also called clusterIP). This address is tied to the lifespan of the Service and will not change while the Service is alive. As service assigns the Pods with their unique IP addresses, as users, we need not keep the track of Pods created when one of the Pod fails. TCP is the default protocol used for Services. A Service can map any incoming port to a targetPort. Each port definition can have the same protocol, or a different one.

Kubernetes networking addresses four concerns[20]:

- Containers within a Pod use networking to communicate via loopback.
- Cluster networking provides communication between different Pods.
- The Service resource allows us to expose an application running in Pods to be reachable from outside the cluster.
- We can also use Services to publish services only for consumption inside the cluster.

Docker uses host-private networking, so containers can talk to other containers only if they are on the same machine. In order for Docker containers to communicate across nodes, there must be allocated ports on the machine's own IP address, which are then forwarded or proxied to the containers. This obviously means that containers must either coordinate which ports they use very carefully, or ports must be allocated dynamically.

Service Topology enables a service to route traffic based upon the Node topology of the cluster. By default, traffic sent to a ClusterIP or NodePort Service may be routed to any backend address for the Service. Kubernetes 1.7 made it possible to route "external" traffic to the Pods running on the same Node that received the traffic.

DNS

Kubernetes creates DNS records for services and pods. Kubernetes DNS schedules a DNS Pod and Service on the cluster and configures the kubelets to tell individual containers to use the DNS Service's IP to resolve DNS names. Every Service defined in the cluster (including the DNS server itself) is assigned a DNS name. By default, a client Pod's DNS search list includes the Pod's own namespace and the cluster's default domain. Pods in a service can be reached by `<namespace>.svc.cluster.local`. In general, a pod has the following DNS resolution: **pod-ip-address.my-namespace.pod.cluster-domain.example**.

Ingress

Ingress may provide load balancing, SSL termination and name-based virtual hosting. Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource[20]. An Ingress with no rules sends all traffic to a single default backend. The defaultBackend is conventionally a configuration option of the Ingress controller

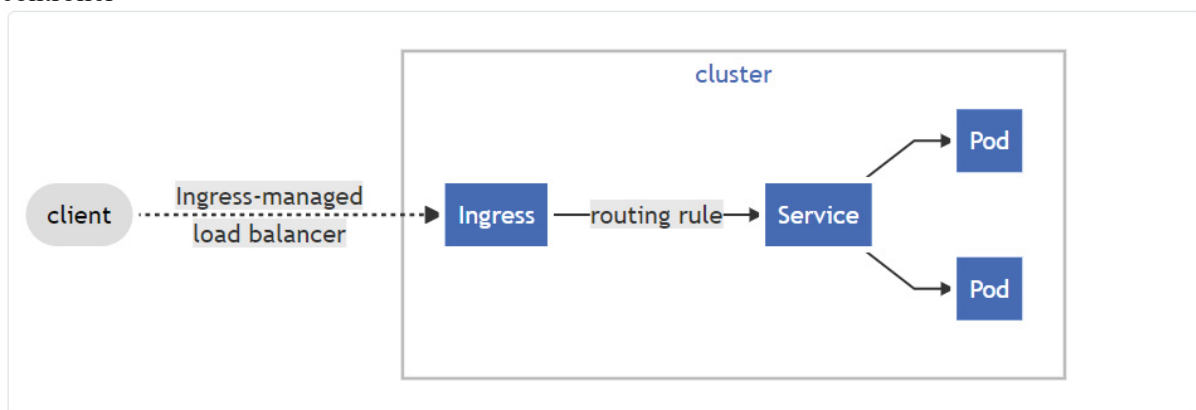


Figure 3.3 Kubernetes Ingress Network

Types of Ingress seen in Kubernetes clusters are below:

- Ingress backed by a single Service
- Simple fanout
- Name based virtual hosting

Securing Services

Before exposing the services, we make sure the communication channel is secure.

- Self-signed certificates for https (unless we already have an identity certificate)
- A server configured to use the certificates
- A secret that makes the certificates accessible to pods

3.2.3 Scaling

The number of nodes in the cluster and the number of pods in deployment to dynamically adjust to the load to meet end user requests. This feature is Scaling. It is inherent in Kubernetes, and it will automatically scale up the cluster as soon as we need it, and scale it back down to save the resources.

Kubernetes has three scalability tools. Two of these, the Horizontal pod autoscaler (HPA) and the Vertical pod autoscaler (VPA), function on the application abstraction layer. The cluster autoscaler works on the infrastructure layer[23].

Kubernetes (like the others) does not provide effective policies for driving the elasticity of latency-sensitive applications. It includes the Horizontal Pod Autoscaler, which uses a threshold-based policy that relies on system-oriented metrics to horizontally scale applications. As such, determining a good scaling threshold is cumbersome, because it requires to identify the relation between a system metric (i.e., utilization) and an application metric (i.e., response time), as well as to know the application bottleneck (e.g., in terms of CPU or memory)[24].

3.2.3.1 Cluster Autoscaler

Autoscalers are responsible for ensuring that the number of pods deployed in the cluster matches the number of pods in its configuration. There is one autoscaler for each service, managing a group of identical, replicated pods which are created from pod templates and can be horizontally scaled. Autoscalers are processes that refer to a pod template and control parameters to scale identical replicas of a pod horizontally, i.e., by increasing or decreasing the number of running copies[21]. Thus, autoscaler facilitate load distribution and increase availability natively within Kubernetes.

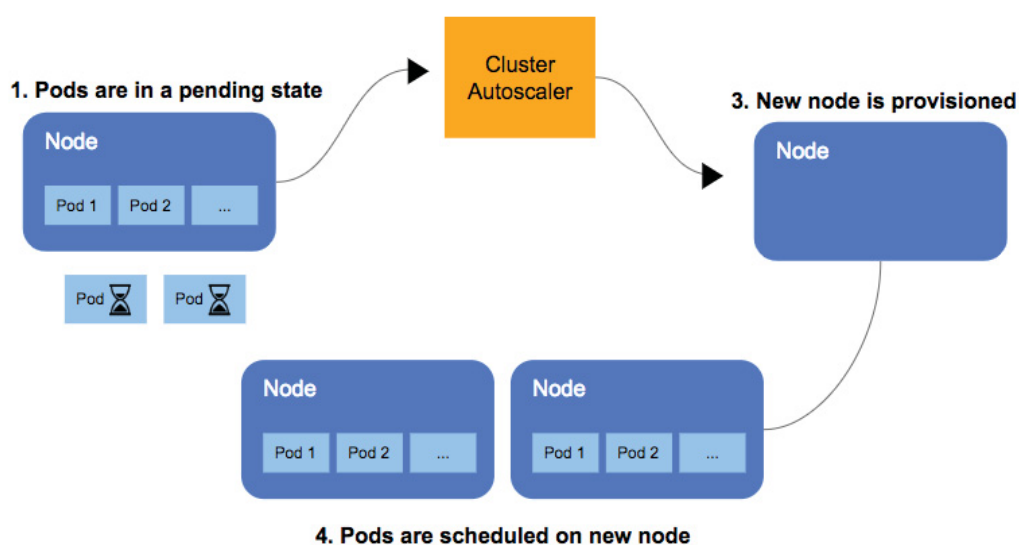


Figure 3.4 Cluster Autoscaler Process

In Kubernetes, we need to use correct version of the Cluster autoscaler and ensure cluster nodes have the same capacity as node groups/instance groups. Since the cluster autoscaler makes scaling decisions based on the scheduling status of pods and the utilization of individual nodes, specifying resource requests is essential for it to function correctly. Kube-system pods by default prevent the cluster autoscaler from scaling down the nodes they are running on. To avoid such situations, we need to specify PodDisruptionBudget on Kube-system pods as well as application pods. Avoid using the Cluster autoscaler with more than 1000 node clusters and also ensure resource availability for the cluster autoscaler pod. The cluster autoscaler has a service level objective (SLO) of 30 seconds latency between the time a pod is marked as unschedulable to the time that it requests a scale-up to the cloud provider. This latency benchmark is for smaller clusters of less than 100 nodes. For larger clusters of up to a 1000 nodes this latency is expected to be around the 60 second mark varying with provider. To avoid this delay and ensure that pods spend as little time as possible in unschedulable state, accomplished using a deployment running pause pods[23].

3.2.3.2 Horizontal Pod Autoscaler(HPA)

The Horizontal Pod Autoscaler is implemented as a Kubernetes API resource and a controller. HPA scales the number of pods in a replication controller, deployment, replica set or stateful set based on CPU Utilization. HPA can also be configured to make scaling decisions based on custom or external metrics. HPA is a great tool to ensure that critical applications are elastic and can scale out to meet increasing demand as well scale down to ensure optimal resource usage.

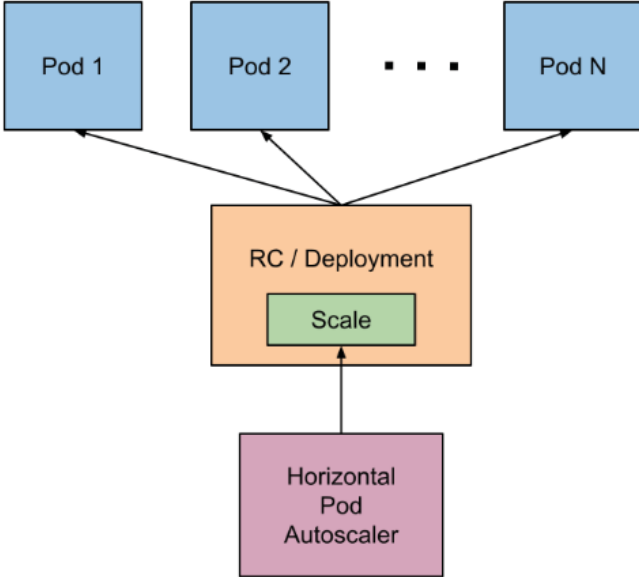


Figure 3.5 Horizontal Pod Autoscaler

To manage a microservice-based application, Kubernetes allows to create multiple Horizontal Pod Autoscaler instances, each carrying out the adaptation of a single microservice deployment[24]. The Kubernetes Horizontal Pod Autoscaling (HPA) offers a reactive-approach where HPA is implemented as a threshold-based reactive controller that automatically adjusts the required resources based on application demand, e.g. through a control loop. Although this decentralized approach can improve scalability, it can also negatively affect the application stability and performance due to frequent and uncoordinated reconfigurations[24].

The HorizontalPodAutoscaler normally fetches metrics from a series of aggregated APIs (metrics.k8s.io, custom.metrics.k8s.io, and external.metrics.k8s.io). The metrics.k8s.io API is usually provided by metrics-server, which needs to be launched separately. The autoscaler accesses

corresponding scalable controllers (such as replication controllers, deployments, and replica sets) by using the scale sub-resource. Scale is an interface that allows anyone to dynamically set the number of replicas and examine each of their current states.

Horizontal Pod Autoscaler, like every API resource, is supported in a standard way by kubectl. We can create a new autoscaler using kubectl create command. It supports scaling during rolling update, great for handling more requests, to get multiple resource metrics [25] and do lot more.

3.2.3.3 Vertical Pod Autoscaler(VPA)

Vertical Pod autoscaling involves adjusting a Pod's CPU and memory requests. Vertical pod Autoscaler exists to automatically set up-to-date resource limits and requests for the containers in their pods. It can both downscale pods that are over-requesting resources and also upscale pods that under-requesting resources based on their usage over time.

Kubernetes clusters work best when all containers of all pods have resource requests, limits for CPU and memory assigned. This effects pod scheduling, lifetime, termination and priority. It is always difficult to know the resources are too low or too high wasting in case of latter. VPA serves best in this situation.

Due to Kubernetes limitations, the only way to modify the resource requests of a running Pod is to recreate the Pod. If we create a Vertical Pod Autoscaler with an updateMode of "Auto", the Vertical Pod Autoscaler evicts a Pod if it needs to change the Pod's resource requests.

The main difference between HPA and VPA is HPA adds or removes both nodes and pods where as VPA changes CPU and memory. Vertical Pod Autoscaler should not be used with the Horizontal Pod Autoscaler (HPA) on CPU or memory unless on custom and external metrics.

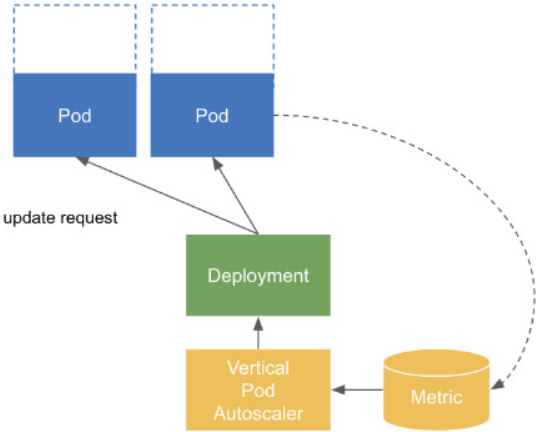


Figure 3.6 Vertical Pod Autoscaler

3.3 Docker Swarm

Docker Swarm was first released in 2016. It has been embedded in the Docker Engine since version 1.12.0. Docker Swarm is a service which allows users to create and manage a cluster of Docker nodes and schedule containers. A swarm is a crew of machines that are functioning Docker and merged into a group. The machines in a swarm can be physical or virtual. Subsequent to joining a swarm, they are alluded to as hubs[52]. Each node of a Docker Swarm is a Docker daemon, and all Docker daemons interact using the Docker API. Services can be deployed and accessed by nodes of same cluster[35].

Swarm extends the Docker container-based software development process on multiple hosts in multiple clouds without any interoperability issue. Docker Swarm-based distributed software development is a newborn approach for the cloud industry. Docker Swarm is also a clustering and scheduling tool, which offers functionalities to turn a group of Docker Systems (Nodes) into a Virtual Docker System [26]. It can provide redundancy in case of Node failure and also workload balancing for containers. Swarm is controlled by Swarm manager, which organize and schedules containers.

Docker Swarm offers an overlay network which exploits the VXLAN tunnel for creating a virtual network among hosts. The Manager node keeps tracks of the state of all nodes in a cluster(in the context of Docker Swarm this service is called discovery and is based on heartbeat mechanism that overlay networking module uses in determining whether a Docker daemon on a remote host in a cluster is still functioning). Docker Swarm allows the fine-grained scalability of part of the service scaling one or more replicated services either up or down to the desired number of replicas[53].

The swarm manager uses ingress load balancing to expose the services we want to make available externally to the swarm. The swarm manager can automatically assign the service a PublishedPort or can be configured for the service. Any unused port can be specified, or swarm manager assigns the service a port in 30000-32767 range by default[28]. External components, such as cloud load balancers, can access the service on the PublishedPort of any node in the cluster whether or not the node is currently running the task for the service. All nodes in the swarm route ingress connections to a running task instance.

Swarm mode has an internal DNS component that automatically assigns each service in the swarm a DNS entry. The swarm manager uses internal load balancing to distribute requests among services within the cluster based upon the DNS name of the service[28].

3.3.1 Features of Docker Swarm

- **Cluster management integrated with Docker Engine:** Use the Docker Engine CLI to create a swarm of Docker Engines where we can deploy application services. We do not need additional orchestration software to create or manage a swarm.
- **Decentralized design:** The key feature of Docker Swarm is fully decentralized, which makes it easy for teams to be able to access and manage the environment. Docker Engine handles any specialization at runtime instead at the time of deployment. This means an entire swarm can be built from a single disk image.
- **Secure by default:** High security is the fundamental requirement for any kind of solution, where in Docker Swarm has that built in. Each node in the swarm enforces TLS mutual authentication and encryption to secure communications between the Manager and all Client nodes within. There is also an option to use self-signed root certificates or certificates from a custom root CA.
- **Load balancing:** It has auto load balancing within the environment that helps us to write out and structure the swarm environment and then allows to convert that swarm environment into a highly scalable infrastructure. We can expose the ports for services to an external load balancer.
- **Scaling:** For each service, we can declare the number of tasks we want to run. When we scale up or down, the swarm manager automatically adapts by adding or removing tasks to maintain the desired state.
- **Rolling updates:** Rollback tasks allows us to be able to rollback environments to previous safe environments. The swarm manager enables us to control the delay between service deployment to different sets of nodes. If anything goes wrong, we can roll back to a previous version of the service.

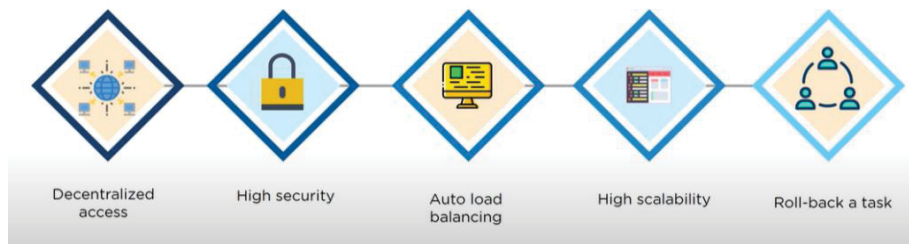


Figure 3.7 Docker Swarm Features

- **Declarative service model:** Docker Engine uses a declarative approach to let us define the desired state of the various services in the application stack. For example, we might describe an application comprised of a web front end service with message queuing services and a database backend.
- **Desired state reconciliation:** The swarm manager node constantly monitors the cluster state and reconciles any differences between the actual state and the expressed desired state. For example, if we set up a service to run 10 replicas of a container, and a worker machine hosting two of those replicas crashes, the manager creates two new replicas to replace the replicas that crashed. The swarm manager assigns the new replicas to workers that are running and available.
- **Multi-host networking:** Overlay network is specified for our services. The swarm manager automatically assigns addresses to the containers on the overlay network when it initializes or updates the application.
- **Service discovery:** Swarm manager nodes assign each service in the swarm a unique DNS name and load balances running containers. Every container running in the swarm can be queried through a DNS server embedded in the swarm[35].

3.3.2 Architecture

A node is an instance of the Docker engine participating in the swarm. There are two types of nodes in Docker Swarm, the manager node and the Worker node. The manager node maintains cluster management tasks and the Worker nodes receive and execute tasks from manager node. Manager node knows the status of all the worker nodes in a cluster. Worker nodes accept tasks sent from manager node. Every worker node has an agent, which reports on the state of the node's tasks to the manager. The worker nodes communicate with the manager node using API over HTTP.

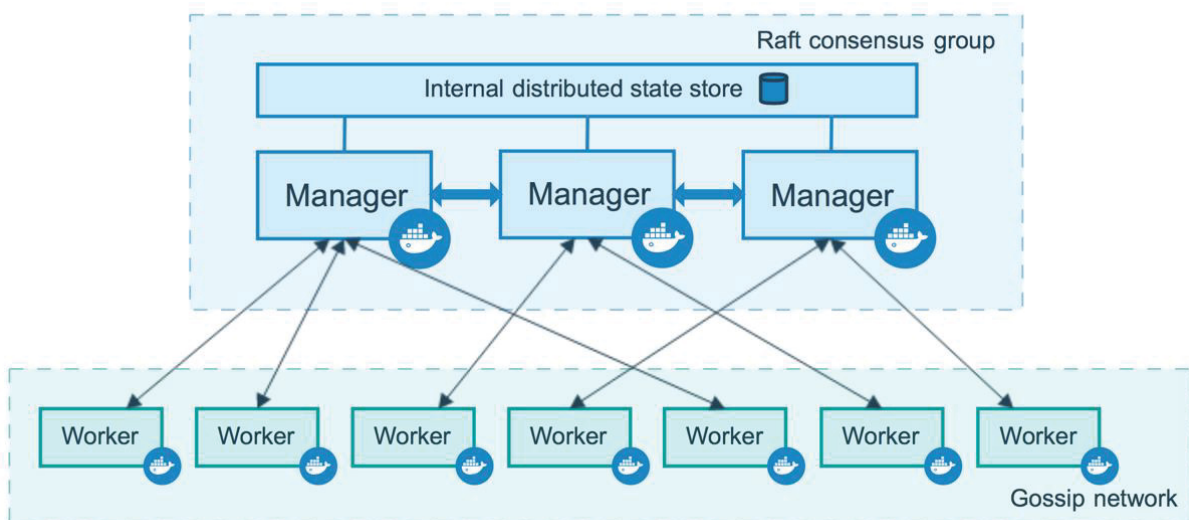


Figure 3.8 Docker Swarm Manager Worker Node Architecture

A Docker Swarm cluster includes the following components and features:
 Raft Consensus group consists of the internal distributed state store in the manager node. Internal Distributed State Store is a built-in key-value store that is used to maintain the cluster state.

Manager Node in Docker Swarm consists of:

- An API that accepts commands and creates a new service based on the definition
- An Orchestrator that reconciles the service definition and creates tasks accordingly
- An Allocator that allocates IP addresses
- A Scheduler that schedules and assigns tasks to worker nodes, and
- A Dispatcher that conducts check-in actions on worker nodes.

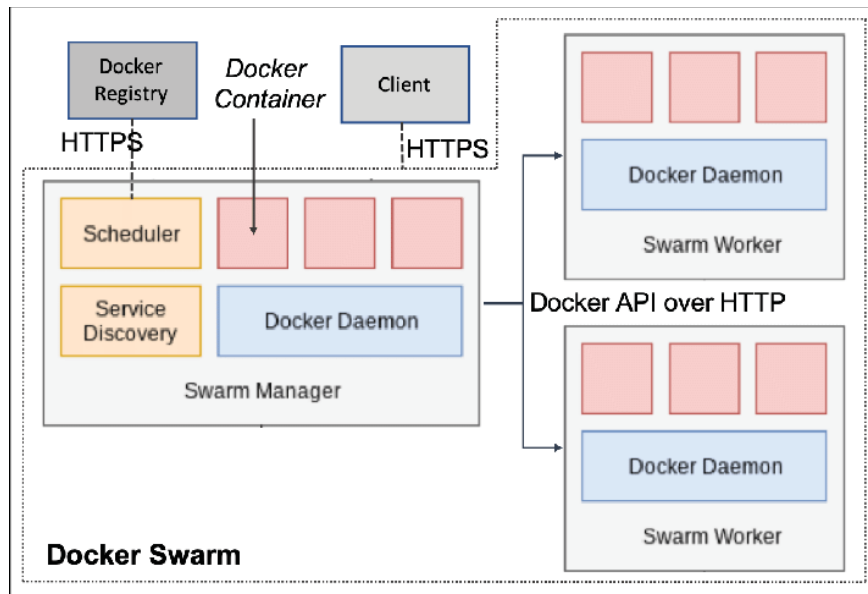


Figure 3.9 Docker Swarm Node internals

When the manager node is promoted as a leader node, it is able to conduct orchestration and management tasks such as:

- receiving the service definition from users.
- replicating the number of tasks based on the service definition and dispatching to worker nodes.
- performing orchestration and cluster management.
- supporting ingress load balancing to expose the services to external, and
- working with the internal distributed state store to maintain the state of the cluster for scheduling and distributing decisions.
- Worker Nodes receive and execute tasks directly from the manager node. Worker nodes also send the status of tasks and a heartbeat to the manager node, to check that the worker node is alive and able to accept tasks.
- A Service consists of one or more replica tasks.
- A Task refers to the combination of a single Docker container and commands of how it will be run.

In terms of fault tolerance, Docker Swam allows multiple manager nodes to co-exist in a cluster to recover from failures without any downtime. When the existing leader is down or not available, the raft consensus group will elect a new leader to conduct orchestration tasks.

3.3.3 Services

In Docker Swarm, containers are launched using services. A service is a group of containers of the same image. Services enables to scale the application. Before we can deploy a service in Docker Swarm, we must have at least one node deployed. In Docker Swarm, services can be deployed and accessed by any node of same cluster. While creating a service, a user has to specify which container image to use. A service is either global or replicated. A global service will run on every Swarm node whereas in a replicated service, the manager node distributes the tasks to specific worker nodes within a swarm.

A service is a description of a task or the state, whereas a task is the work that need to be done. Task is the atomic scheduling unit of swarm. Docker enables a user to create services, which can start tasks. When a task is assigned to a node, it cannot be assigned to another node. It can only run on the assigned node or fail. It is possible to have multiple manager nodes on Swarm, but there will be only one Primary Manager node, which gets elected by the other Manager nodes which are considered as the Secondary Managers. Secondary Managers are similar in concept to Worker nodes in which they have the capability of a manager, but they are dependent on that single Primary manager to be able to provide the right instructions, services and tasks to the entire Swarm environment. By default, manager nodes also run services as worker nodes, but we can configure them to run manager tasks exclusively and be manager-only nodes.

Stack service

Docker stack is a command that is embedded into the Docker CLI. It lets us manage a cluster of Docker containers through Docker Swarm. All Swarm objects can and should be described in manifests called stack files. These simple YAML files describe all the components and configurations, everything we need to create our application in a running state; we can check it into version control and share it among others, allowing to distribute the applications to other clusters and also destroy in any Swarm environment. Docker Compose can also be used to define and run containers via Swarm service stacks.

Kubernetes Services and Swarm Services refer to different functionalities. The two orchestrators mean very different things by the term 'service'. In Swarm, a service provides both scheduling and networking facilities, creating containers and providing tools for routing traffic to them. In Kubernetes, scheduling and networking are handled separately, deployments handle the scheduling of containers as pods, while services are responsible only for adding networking features to those pods[36].

3.3.4 Networking

Docker uses embedded DNS to provide service discovery for containers running on a single Docker Engine and tasks running in a Docker Swarm. Docker Engine has an internal DNS server that provides name resolution to all of the containers on the host in user-defined bridge, overlay, and MACVLAN networks[31].

As mentioned, Networking is handled by services in Docker Swarm when stack files are deployed. We can also create a network using commands and can connect all the containers to that network. One of the reasons Docker containers and services are so powerful is that we can connect them together or connect them to non-Docker workloads. Docker containers and services do not even need to be aware that they are deployed on Docker, or whether their peers are also Docker workloads or not. It is not confined to host definition. Below are the few types of networks in Docker.

- **User-defined bridge networks** are best when we need multiple containers to communicate on the same Docker host.

- **Host networks** are best when the network stack should not be isolated from the Docker host, but we want other aspects of the container to be isolated.
- **Overlay networks** are best when we need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.
- **Macvlan networks** are best when we are migrating from a VM setup or need the containers to look like physical hosts on the network, each with a unique MAC address.
- **Third-party network plugins** allow us to integrate Docker with specialized network stacks.

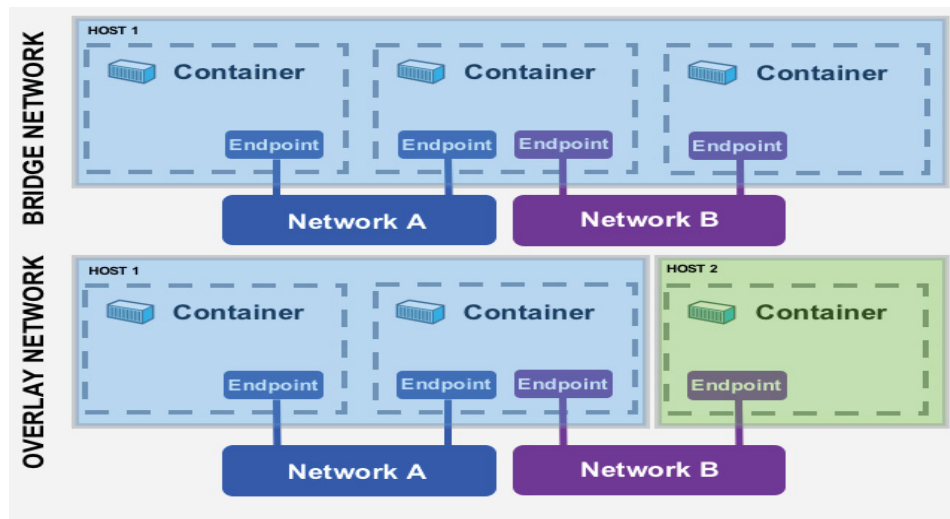


Figure 3.10 Docker Bridge and Overlay Network

Overlay networks are being used in Swarm. There are different kinds of overlay networks used in different scenarios[27].

- Default overlay network sets up automatically when we initialize or join a swarm. This network is not the best choice for production systems.
- User-defined overlay networks is a custom overlay network to connect services. This is recommended for services running in production.
- Overlay network for standalone containers is to communicate between standalone containers on different Docker daemons.
- Attachable overlay network enables communicate between a standalone container and a swarm service

3.3.5 Scaling

Docker CLI can be used to scale the number of containers in the service. Unlike Kubernetes, there is no automatic way of scaling the containers. A command needs to be passed for swarm manager to launch the containers based on request of resources. Docker Swarm is effective to keep the containers alive with specified number of replicas.

We can use different tools to monitor the Docker Swarm services, depending on the values we can either scale the resources up or down. Few such monitoring tools are cadvisor and node-exporter using Prometheus, Orbiter, dockerflow monitor[29].

According to [30], the self-adaptation of docker services to scale up involves many steps. The tools used in the setup are as follows

- Prometheus: Scrapes metrics and fires alerts when certain thresholds are reached.
- Docker Flow Monitor: It extends Prometheus with capability to auto-configure itself.

- Alertmanager: Receives alerts from Prometheus and forwards them to some other service depending on matching routes.
- Jenkins: Executes scheduled or triggered jobs. This will be used as the engine that will scale a service.
- Docker Flow Proxy: It extends HAProxy with capability to auto-configure itself.
- Docker Flow Swarm Listener: Listens to Swarm events and sends notifications when a service is created or updated. It will be used to send notifications to *Docker Flow Monitor* and *Docker Flow Proxy*.

3.3.6 Advantages

Docker Swarm is a container orchestration tool, meaning that it allows the user to manage multiple containers deployed across multiple host machines. One of the key benefits associated with the operation of a Docker Swarm is the high level of availability offered for applications. Another key advantages of swarm services over standalone containers are that we can modify a service's configuration, including the networks and volumes it is connected to, without the need to manually restart the service. Docker will update the configuration, stop the service tasks with the out-of-date configuration, and create new ones matching the desired configuration[28].

3.4 WebRTC

Web Real-Time Communication (WebRTC) is a collection of standards, protocols, and JavaScript APIs, the combination of which enables peer-to-peer audio, video, and data sharing between browsers (peers). Instead of relying on third-party plug-ins or proprietary software, WebRTC turns real-time communication into a standard feature that any web application can run on a simple JavaScript API[40].

With WebRTC, we can add real-time communication capabilities to the application that works on top of an open standard. The technology is available on all modern browsers as well as on native clients for all major platforms [18]. Support for enabling real-time communication (RTC) in the Web has gained momentum with the two main Internet standardization bodies – the IETF and W3C[41]. The WebRTC framework provides the building blocks from which web and app developers can seamlessly add video chat to a range of applications, including tele-education and tele-health, entertainment and gaming, professional and workforce collaboration. WebRTC is massively deployed as a communication platform and powers video conferences and collaboration systems across all major browsers, both on desktop and mobile. Billions of users, from startups to Web-scale companies, in commercial products and open-source projects, WebRTC has vastly expanded the ability to deploy real-time interaction solutions to customers and users[42].

Browser-to-browser communication is a revolutionary approach to Web-based communication because it allows P2P (in which each peer is a browser) data communication to enter the Web application arena for the first time. Real-time flow of data streams across the network is direct with no further intermediaries along the path.

The WebRTC standard covers two different technologies: media capture devices and peer-to-peer connectivity. Media capture devices includes video cameras and microphones, but also screen capturing "devices". For cameras and microphones, we use `navigator.mediaDevices.getUserMedia()` API to capture Media Streams and for screen recording, we use `navigator.mediaDevices.getDisplayMedia()` API instead. The peer-to-peer connectivity is handled by the `RTCPeerConnection` interface. This is the central point for establishing and controlling the connection between two peers in WebRTC[18]. At present, WebRTC supports two peers at a time.

The codecs and protocols used by WebRTC do a huge amount of work to make real-time communication possible, even over unreliable networks[44] such as Packet-loss concealment, Echo cancellation, Bandwidth adaptivity, Dynamic jitter buffering, Automatic gain control, Noise reduction and suppression, Image-cleaning.

3.4.1 Architecture

RTC's architectural model is the browser RTC trapezoid, which lets the media path flow directly between browsers without any intervening servers. The signaling path crosses servers that can modify, translate, or manage signals as needed.

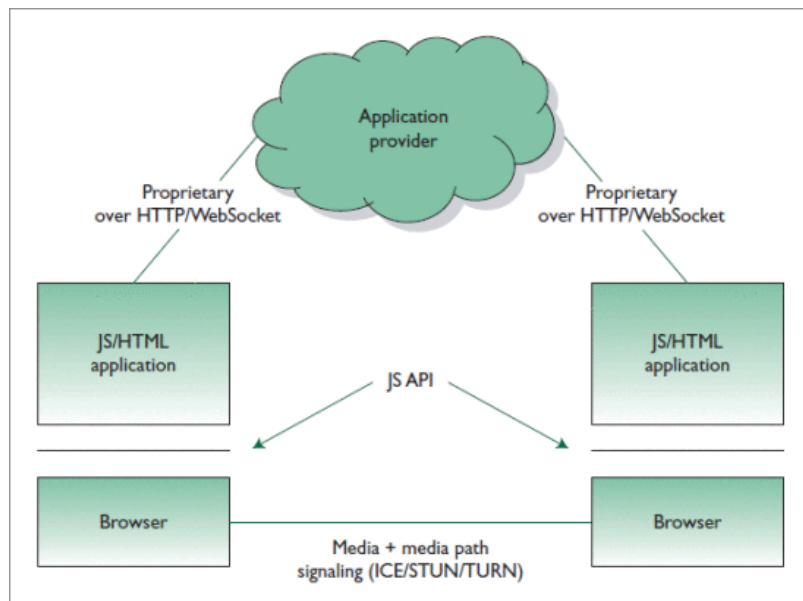


Figure 3.11 WebRTC Architectural Model

The client-side Web applications (typically written in a mix of HTML and JavaScript) interact with Web browsers through the WebRTC API, letting them properly exploit and control browser functions and interact with browsers themselves in both a proactive (for instance, to query browser capabilities) and reactive (to receive browser-generated notifications) way[41]. The application-browser API must thus provide a wide set of functions, such as connection management (in a P2P fashion), encoding/decoding capabilities, negotiation, selection and control, media control, firewall, and network address translation (NAT) traversal.

Communication in real-time audio and video call between two browsers might involve direct media streams, with the media path negotiated and instantiated through a complex sequence of interactions involving the following entities[41]:

- the caller browser and the caller JavaScript application (through the JavaScript API)
- the caller JavaScript application and the application provider (Web server)
- the application provider and the callee JavaScript application
- the callee JavaScript application and the callee browser (through the application-browser JavaScript API).

The simplified process of WebRTC connection involves the following

- both clients obtain their local media streams
- once the stream is obtained, each client connects to the signaling server

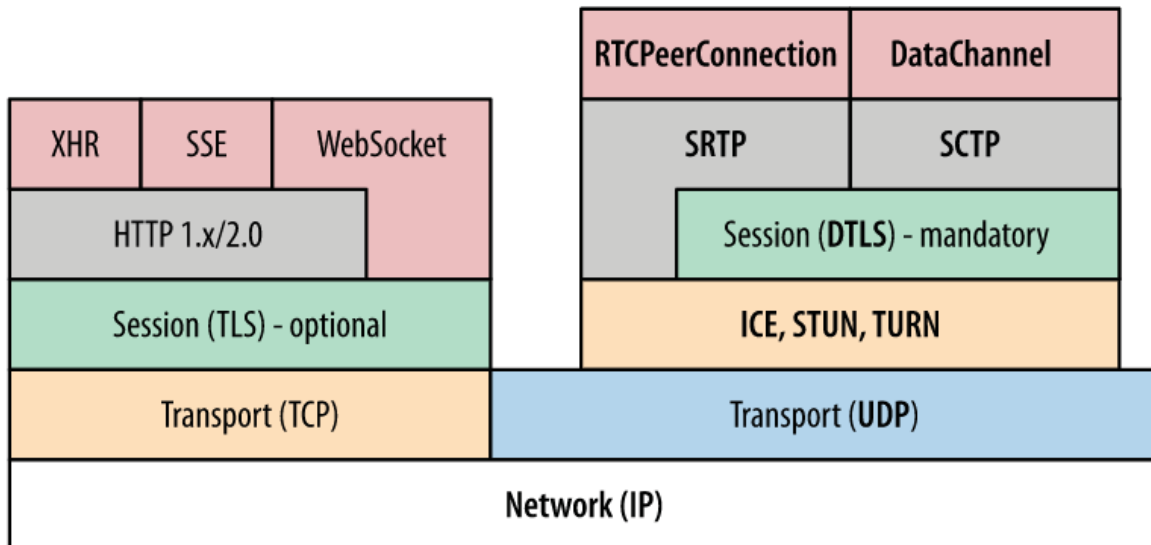


Figure 3.13 WebRTC Network stack

- ICE: Interactive Connectivity Establishment (RFC 5245)
 - STUN: Session Traversal Utilities for NAT (RFC 5389)
 - TURN: Traversal Using Relays around NAT (RFC 5766)
- SDP: Session Description Protocol (RFC 4566)
- DTLS: Datagram Transport Layer Security (RFC 6347)
- SCTP: Stream Control Transport Protocol (RFC 4960)
- SRTP: Secure Real-Time Transport Protocol (RFC 3711)

ICE, STUN, and TURN are necessary to establish and maintain a peer-to-peer connection over UDP. DTLS is used to secure all data transfers between peers; encryption is a mandatory feature of WebRTC. Finally, SCTP and SRTP are the application protocols used to multiplex the different streams, provide congestion and flow control, and provide partially reliable delivery and other additional services on top of UDP[40].

3.4.3 API

The W3C WebRTC API lets a JavaScript application exploit the novel browser's real-time capabilities. It requires the browser core to provide the functionality needed to establish the necessary audio, video, and data channels. The media and data streams will always be encrypted.

To acquire and communicate streaming data, WebRTC implements the following APIs:

- `MediaStream` gets access to data streams, such as from the user's camera and microphone.
- `RTCPeerConnection` enables audio or video calling with facilities for encryption and bandwidth management.
- `RTCDataChannel` enables peer-to-peer communication of generic data.

MediaStream

A `MediaStream` is an abstract representation of an actual data stream of audio or video. It serves as a handle for managing actions on the media stream such as displaying the stream's content, recording it, or sending it to a remote peer. A `MediaStream` can be extended to represent a stream that either comes from (remote stream) or is sent to (local stream) a remote node. A `LocalMediaStream` represents a media stream from a local media-capture device (such as a webcam or microphone).

To create and use a `LocalMediaStream`, the Web application must request access from the user via the `getUserMedia()` function. The application specifies the type of media, audio or video to which it requires access. The device selector in the browser interface grants or denies access. Once the application is done, it can revoke its own access by calling the `stop()` function on the `LocalMediaStream`[41].

The `getUserMedia()` API allows us to specify a list of mandatory and optional constraints to match the needs of the application in case of constrained capabilities of the input source. The media is automatically optimized, encoded, and decoded by the WebRTC audio and video engines and is then routed to one or more outputs. The audio and video engines take care of all the signal processing, adjust to network jitter, latency delays and more[40].

RTCPeerConnection

A `RTCPeerConnection` API enables two users to communicate directly, browser-to-browser. It is responsible for managing the full life cycle of each peer-to-peer connection. It then represents an association with a remote peer, which is usually another instance of the same JavaScript application running at the remote end. Communications are coordinated via a signaling channel provided by scripting code in the page via the Web server using `XMLHttpRequest` or `WebSocket`. Once the calling browser establishes a peer connection, it can send `MediaStream` objects directly to the remote browser.

The peer-connection mechanism uses the ICE protocol along with the Session Traversal Utilities for NAT (STUN) and Traversal Using Relays around NAT (TURN) servers to let UDP-based media streams traverse NAT boxes and firewalls. When peers are on the same network, they can reach each other without STUN or TURN servers. ICE lets browsers discover enough information about the topology of the network on which they are deployed to find the best exploitable communication path. Using ICE also provides a security measure because it prevents untrusted webpages and applications from sending data to hosts that are not expecting to receive it[41].

The remote host feeds each signaling message into the receiving `PeerConnection` on arrival. The APIs send signaling messages that most applications will treat as opaque blobs, but which the Web application must transfer securely and efficiently to the other peer via the Web server. `RTCPeerConnection` encapsulates all the connection setup, management, and state within a single interface[40].

RTCDataChannel

The `DataChannel` provides a generic transport service that lets Web browsers exchange generic data in a bidirectional, P2P fashion. Within the IETF, standardization work has reached a general consensus on using the Stream Control Transmission Protocol (SCTP) encapsulated in DTLS to handle nonmedia data types. Encapsulating “SCTP over DTLS over ICE over UDP” provides a NAT traversal solution together with confidentiality, source authentication, and integrity-protected transfers. Moreover, this solution lets the data transport interwork smoothly with parallel media transports, and both can share a single transport-layer port number. The IETF chose SCTP because it natively supports multiple streams with reliable, unreliable, and partially reliable delivery modes. It allows applications to open several independent streams (up to 65,536 in each direction) within an SCTP association toward a peering SCTP endpoint. Each stream actually represents a unidirectional logical channel, providing in-sequence delivery[41].

An application can send a message sequence either ordered or unordered. The message delivery order is preserved only for all ordered messages sent on the same stream. However, the `DataChannel` API is bidirectional, which means that each `DataChannel` bundles an incoming and an outgoing SCTP stream. An application sets up a data channel (that is, creates the SCTP association) when the `CreateDataChannel()` function is called for the first time on an instantiated `PeerConnection` object. Each subsequent call to the `CreateDataChannel()` function just

creates a new DataChannel within the existing SCTP association[41]. Each configuration of the channel has its own performance characteristics and limitations[40].

Sending a message on a RTCDataChannel is done by calling the send() function with the data we want to send. The data parameter for this function can be either a string, a Blob, an ArrayBuffer or and ArrayBufferView. The remote peer will receive messages sent on a RTCDataChannel by listening on the message event[18].

3.4.4 Signaling

Since its inception, the general idea behind WebRTC's design has been to fully specify how to control the media plane while leaving the signaling plane to the application layer as much as possible[41]. The WebRTC specification includes APIs for communicating with an ICE (Internet Connectivity Establishment) Server, but the signaling component is not part of it[18]. A connection is established through a discovery and negotiation process called signaling. Signaling involves both devices connecting to a third, mutually agreed-upon server. Through this third server, the two devices can locate one another, and exchange negotiation messages[43]. The RTCPeerConnection object is responsible for this[45]. As soon as devices know how to find each other over the Internet, they start exchanging data about which protocols and codecs each device supports. There is a minimum exposure of private information.

The rationale is that different applications might prefer to use different standardized signaling protocols (such as SIP or XMPP) or even something custom which may consist of one or more servers to communicate messages. This allows interoperability with a variety of other signaling protocols powering existing communications infrastructure[40]. The important information that browsers must exchange is the multimedia session description, which specifies the transport (and Interactive Connectivity Establishment [ICE]) information as well as the media type, format, and all associated media configuration parameters necessary to establish the media path[41].



Figure 3.14 WebRTC Signaling

Session Description Protocol

The original idea to exchange session description information in the form of Session Description Protocol (SDP) blobs[41]. SDP does not deliver any media instead lists the properties of connection. WebRTC applications do not deal with SDP directly. The JavaScript Session Establishment Protocol (JSEP) abstracts all the inner workings of SDP behind a few simple method calls on the RTCPeerConnection object.

Caller initiates the call, requests audio/video streams from browser and creates a SDP offer after registering with RTCPeerConnection object. Now this SDP offer is sent to callee(remote peer) over signaling channel as local description. Once the offer is received by remote peer, he sets local description as the "remote description" of the session, registers his own streams with his own RTCPeerConnection

object, generates the "answer" SDP description, and sets it as the "local description" of the session. Now this is sent to our Caller, and he sets the received SDP answer as the "remote description" of his original session. With that, once the SDP session descriptions have been exchanged via the signaling channel, both parties have now negotiated the type of streams to be exchanged, and their settings.

Interactive Connectivity Establishment (ICE)

For the communication to establish fully, we need to collect the ICE candidates at each peer and transfer (over the signaling channel) to the other peer. Before two peers can communicate using WebRTC, they need to exchange connectivity information.

In a trivial case, where both peers are located on the same internal network, and there are no firewalls or NATs between them. To establish the connection, each peer can simply query its operating system for its IP address (or multiple, if there are multiple network interfaces), append the provided IP and port tuples to the generated SDP strings, and forward it to the other peer. Once the SDP exchange is complete, both peers can initiate a direct peer-to-peer connection[40].

Since the network conditions can vary depending on a number of factors, an external service is usually used for discovering the possible candidates for connecting to a peer. This service is called ICE and is using either a STUN or a TURN server. STUN stands for Session Traversal Utilities for NAT and is usually used indirectly in most WebRTC applications[18]. TURN (Traversal Using Relay NAT) is the more advanced solution that incorporates the STUN protocols and most commercial WebRTC based services use a TURN server for establishing connections between peers. The WebRTC API supports both STUN and TURN directly, and it is gathered under the more complete term Internet Connectivity Establishment.

When creating a WebRTC connection, we usually provide one or several ICE servers in the configuration for the `RTCPeerConnection` object. Once a `RTCPeerConnection` object is created, the underlying framework uses the provided ICE servers to gather candidates for connectivity establishment (ICE candidates). The event `icegatheringstatechange` on `RTCPeerConnection` signals in what state the ICE gathering is (new, gathering or complete)[18].

While it is possible for a peer to wait until the ICE gathering is complete, it is usually much more efficient to use a "trickle ice" technique and transmit each ICE candidate to the remote peer as it gets discovered. This will significantly reduce the setup time for the peer connectivity and allow a video call to get started with less delays. To gather ICE candidates, we add a listener for the `icecandidate` event. The `RTCPeerConnectionIceEvent` emitted on that listener will contain candidate property that represents a new candidate that should be sent to the remote peer.

Once ICE candidates are being received, we should expect the state for our peer connection will eventually change to a connected state. To detect this, we add a listener to our `RTCPeerConnection` where we listen for `connectionstatechange` events[18].

A WebRTC session may require multiple streams for delivering audio, video, and application data. As a result, a successful connection is one that is able to establish connectivity for all the requested streams. Further, due to the unreliable nature of peer-to-peer connectivity, there are no guarantees that once the connection is established that it will stay that way: the connection may periodically flip between connected and disconnected states while the ICE agent attempts to find the best possible path to re-establish connectivity[40].

3.5 Kurento Media Server

Kurento Media Server is an open source WebRTC Media Server providing a toolbox of capabilities which include group communications, recording, routing, transcoding and mixing. Kurento supports a

large number of media protocols such as WebRTC, plain RTP, RTSP or HTTP and bunch of codecs including VP8, VP9, H.264, H.263, OPUS, Speex, PCM or AMR. Kurento Media Server is based on a modular architecture, which makes it possible for developers to extend and customize its native capabilities with advanced media processing features such as computer vision, augmented reality or speech analysis. Kurento is ideal for WWW developers who find natural programming with its Java and JavaScript APIs following the traditional three-tiered WWW development model[33]. WebRTC without media server is enough for creating basic applications but features such as group communications, media stream recording, media broadcasting or media transcoding are difficult to implement on top of it. For this reason, most interesting applications require using a media server. Most available RTC media servers are designed and optimized to provide one of the following capabilities such as Group Communication Capabilities, Media Archiving Capabilities and Media Bridging[46].

- Group Communication Capabilities enable groups of users to communicate synchronously. Most group videoconferencing services require them.
- Media Archiving Capabilities are related to the ability to record multimedia streams into media repositories and recover them later for visualization.
- Media Bridging provides media interoperability among different network domains having incompatible media formats or protocols.

Kurento is designed based on the following main principles[48]:

- Distribution of Media and Application Services
- Suitable for the Cloud
- Media Pipelines
- Application development
- End-to-End Communication Capability
- Fully Processable Media Streams
- Modular Processing of Media
- Auditable Processing
- Seamless IMS integration
- Transparent Media Adaptation Layer

3.5.1 Architecture

The architecture of Kurento is split into media and signaling planes. The former is based on the JBoss/Mobicents Java EE stack, while the latter has been built on top of the GStreamer media pipeline framework. Both communicate using Thrift RPCs for exchanging media control information. This architecture exposes the powerful media capabilities of GStreamer through the flexible and interoperable framework provided by Java EE technologies[47]. The conceptual representation of the Kurento architecture can be seen on Figure 2, where the separation between the Kurento Signaling Server (KSS) and the Kurento Media Server (KMS) can be observed.

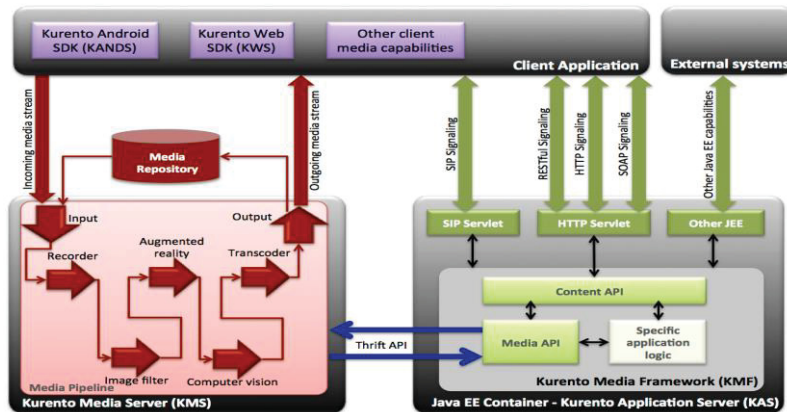


Figure 3.15 Kurento Architecture

Kurento Application Server (KAS): This consists on a Java EE compatible container (such as JBoss, Tomcat or equivalent) on top of which the Kurento Media Framework (KMF) is deployed. KMF has been conceived to provide Java developers a rich toolbox suitable for satisfying common requirements emerging in multimedia services. KMF comprises two sub-APIs[34].

Content API: This is a signaling API, meaning that it is in charge of managing control communications with the client. It provides developers the ability to specify the application logic that executes when specific events occur (e.g., when the client wants to establish a session, when the media exchange starts, when media exchange concludes, when errors arise, etc.) Thanks to this API, developers can customize aspects such as authentication, permissions, accounting, etc. For all this, developers will have access to the rich environment exposed by the Java platform.

Media API: This is a media control API. It consists of stubs that make possible controlling the lifecycle and behavior of Kurento Media Server (KMS) media capabilities. The Media API is an abstract API so that most of the complex low-level details (garbage collection, format and codecs, etc.) are hidden.

Kurento Media Server (KMS): KMS is the core of the Kurento infrastructure. It is a media server written on top of GStreamer, which offers different types of media capabilities such as media reception and delivery through different protocols and formats[34].

Kurento client-side APIs: These include Kurento Web SDK (KWS) and Kurento Android SDK (KANDS). Both of them make possible to access Kurento infrastructure capabilities in a simple and seamless manner[34].

4 RESEARCH METHODOLOGY

This section describes about the methodologies adopted to achieve the objectives of the thesis.

- Literature review about the related works to gain sufficient knowledge about Kurento, Kubernetes, Docker Swarm.
- Setup of Docker and Kubernetes in the VM.
- Pull docker images and configure the video conferencing application and Kurento image.
- Start Kubernetes cluster and start video call application.
- Launch Docker Swarm cluster and start video call application.
- Conduct tests to extract the performance metrics CPU Utilization, Memory Utilization, Network Traffic, Packet Loss and determine container overhead.
- Statistical analysis of the results obtained.

Based on the literature review and reviewing the other research methodologies, quantitative method is adapted in this thesis. Quantitative methods and surveys help to gain numerical outputs that help in all the research. Results can be formed easily without explaining much in the thesis with the help of numbers. We are comparing performance of Kubernetes with Docker Swarm and Native container for WebRTC, so analysis of resource consumption will be in place. To understand which performs better we need to understand which utilizes resources in a better way. So, quantitative(experimental) methodology was selected to calculate, compare and analyse metrics like CPU Utilization, Memory Utilization, Network Traffic and Packet Loss.

Based on the experimentation, the expected outcomes are as follows:

- Analysis of Web based Real time communication server(Kurento) in different container technologies.
- A recommendation of which container technology to use with real time communication for faster response.
- Performance overhead is expected to raise when the application is scaled. The Memory Utilization, CPU Utilization, Network Traffic and Packet Loss are expected to grow.
- Kubernetes has better Memory Utilization and CPU Utilization than Docker Swarm.

4.1 Kurento Video Conferencing Application

4.1.1 Multiparty Architectures

Kurento has number of ways to connect to peers. The application with number of users connecting to the call plays a significant role in determining the best-suited orchestration tool. One-to-One connections are easy to manage and deploy: the peers talk directly to each other, and no further optimization is required. However, extending the same strategy to an N-way call, where each peer is responsible for connecting to every other party (a mesh network) would result in $N-1$ connections for each peer, and a total of $N*(N-1)$ connections.

Kurento framework is used to build different types of WebRTC and multimedia applications such as one to one call, one to many(broadcast), loopback, Players.

The topologies selected for this thesis are One-to-One video call application where each user can connect with only one another user. This type of application is considered as it is so common in the real world communication. Another being One-to-Many application, where one user presents or streams the data and other N users just view it. This topology represents the live streaming of data in real time.

One to One call

One-to-One call is a two-party streaming application which is simple and efficient to deploy, whereas multiparty architectures require a lot more thought and planning. As much as WebRTC is about enabling direct peer-to-peer communication, it is also a catalyst for a wide variety of services, which will help us make it more efficient.

When A and B are using the application, the media flow goes this way: The video camera stream of user A is sent to the Kurento Media Server, which sends it to user B. In the same way, B sends to Kurento Media Server, which forwards it to A. This means that KMS is providing a B2B (back-to-back) call service.

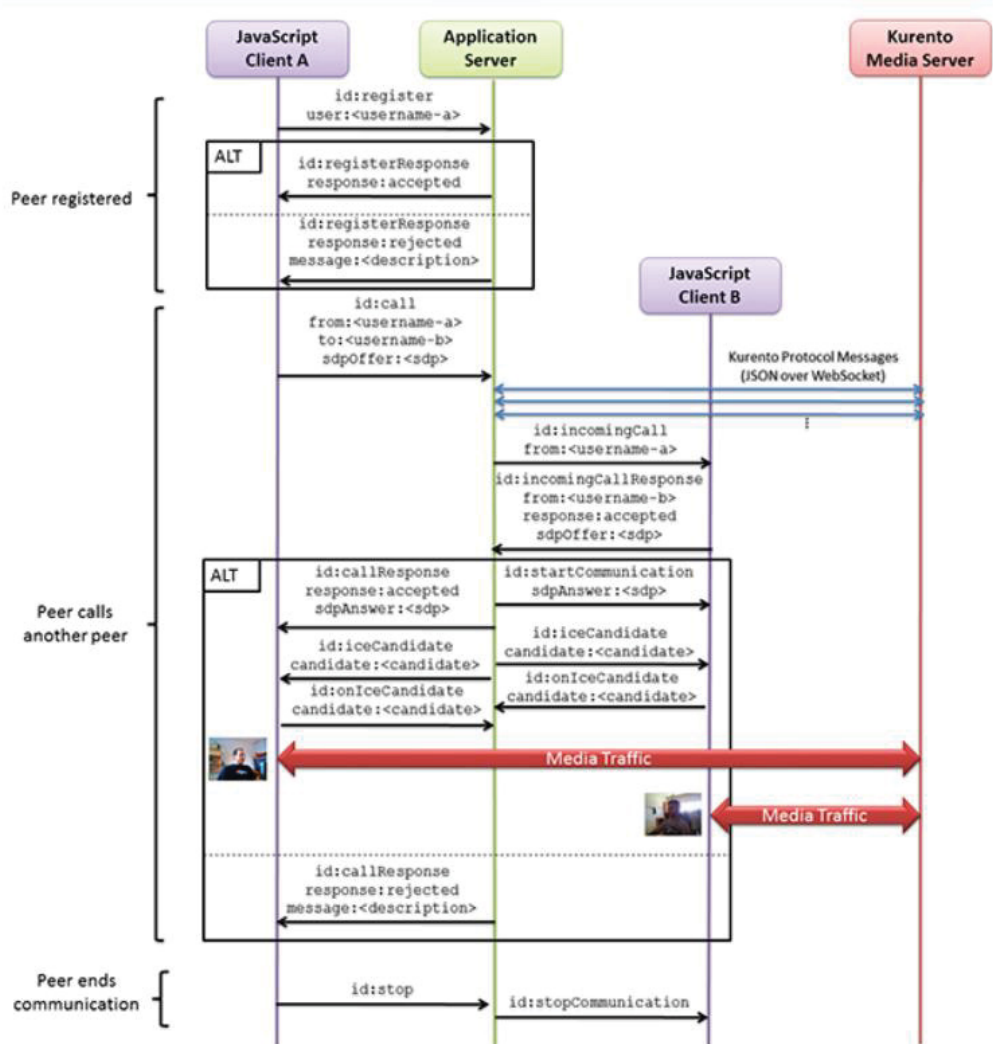


Figure 4.1 One-to-One video call signaling protocol

One to many call (one-N)

One-to-Many call has a Presenter and N viewers who streams the presenters video feed. When A is presenting and B, C are using the application, the media flow goes this way: The video camera stream of user A is sent to the Kurento Media Server, which sends it to user B. Same stream is also sent to C. This way whenever a new connection is arrived, Kurento server transmits the same stream to that connection.

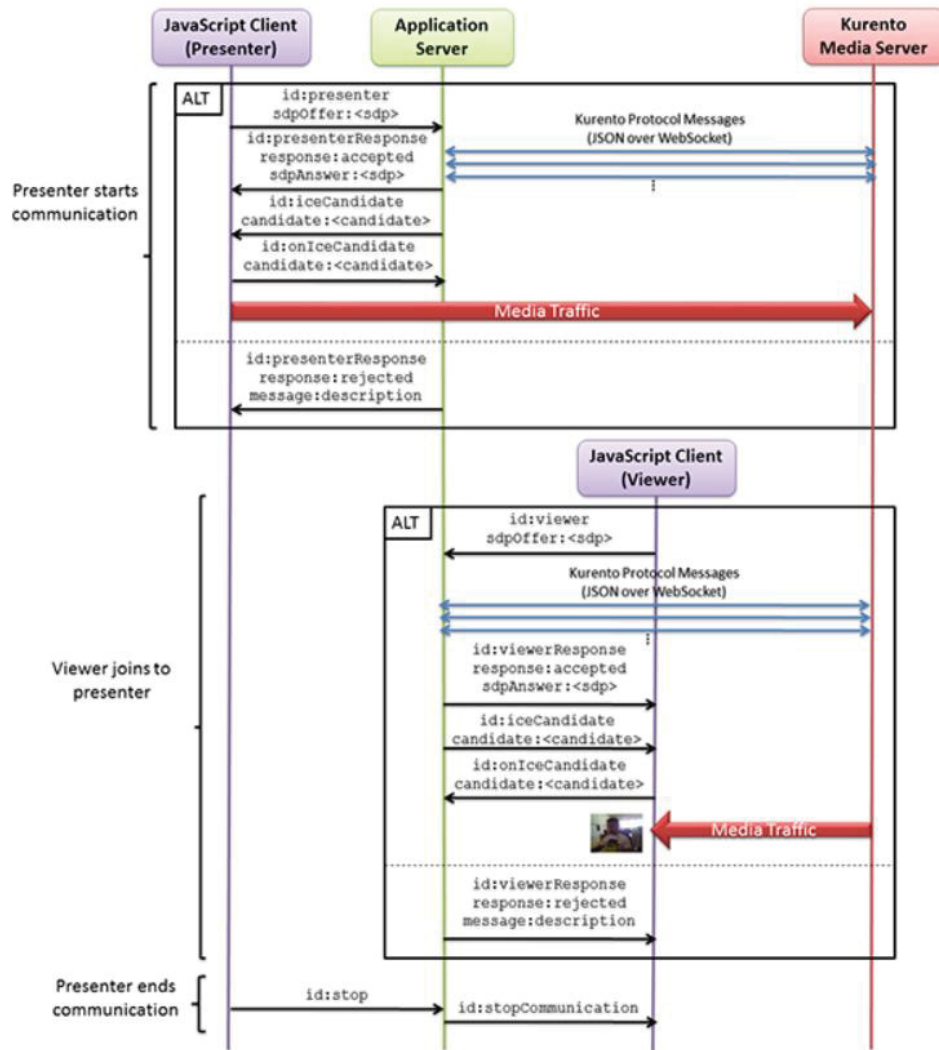


Figure 4.2 One-to-Many video call signaling protocol

4.1.2 Initial application development and Risk State

As we are aware, signalling in WebRTC has no one standard. So, we tried to develop the application from scratch using the WebRTC APIs and Kurento tutorials in Python. For this, PyforKurento module is used. We started to implement it with Django framework(views), but as ICE candidates are exchanged asynchronously, we switched to Django channels. A chat application using Channels was built and tested in Minikube. We encountered a problem where Minikube was unable to recognise the images built(using Docker compose). Soon, we realised the PyforKurento module is yet in the development stage, and we wouldn't be successful with own application development.

4.1.3 Alternative chosen

Therefore, we opted for application server in NodeJs provided as a part of Kurento media server tutorials, modifying it as per requirements and enabling to reach the Kurento server running in a separate container. Also, we implemented the Kubernetes cluster with kubeadm instead of Minikube. Implementing Kubernetes cluster in this method is more realistic than minikube.

4.2 Performance Monitoring Tools

Performance Monitoring tools monitors the performance of various software applications, network and many other devices and more. Performance Monitoring includes individual web requests, transactions, CPU and Memory usage, latency, throughput, Application error, Network Traffic, etc. Few monitoring tools studied as a part of literature review are Kurento Inspector, Datadog, Signoz and Prometheus.

Kurento Inspector or the KMS-Inspector is a tool that makes possible to monitor the internal behavior of Kurento Media Server from our browser. We can visualize the pipelines and media elements and extract QoS metrics from them. It is one of the tools for debugging the Kurento Media Server applications. But KMS-Inspector does not offer integration features as other tools like Prometheus.

Datadog is founded in 2010, offers a look at performance from a comprehensive dashboard. An all-in-one monitoring service, Datadog can monitor, troubleshoot, and optimize application and network performance across the stack – hybrid cloud applications, servers, databases, and applications. It can monitor containers, cloud instances, on-premises, and hybrid architectures. It has support for popular web frameworks (Django, Ruby on Rails, Gin, and Spring), Kubernetes, Docker etc. It has very good features, but it is not an open-source tool. Datadog is planning for open-source tracing libraries along with OpenTelemetry. Considering the factors, we decided to move to other alternatives.

Signoz is an open-source alternative for Datadog, SigNoz uses industry proven Kafka and Druid to power highly scalable data ingestion and real-time data analysis. Durid is a real-time analytics database. It has good support for Docker Swarm but not much for Kubernetes. So, we dropped the idea of using the tool.

Then we came across a leading open-source tool for monitoring the metrics in both Kubernetes and Docker Swarm, Prometheus. In [2], Prometheus was used to measure the docker performance and proven a reliable solution for our comparison. It also has Grafana integration, a powerful visualization tool to look at the metrics.

4.2.1 Prometheus

Prometheus[32] is an open-source system monitoring and alerting toolkit originally built at SoundCloud inception in 2012. It is now a standalone open-source project and joined the Cloud Native Computing Foundation in 2016 as the second hosted project, after Kubernetes.

Prometheus's main features are[32]:

- a multi-dimensional data model with time series data identified by metric name and key/value pairs
- PromQL, a flexible query language to leverage this dimensionality
- no reliance on distributed storage; single server nodes are autonomous
- time series collection happens via a pull model over HTTP
- pushing time series is supported via an intermediary gateway
- targets are discovered via service discovery or static configuration
- multiple modes of graphing and dashboarding support

The Prometheus ecosystem consists of multiple components, many of which are optional:

- the main Prometheus server which scrapes and stores time series data
- client libraries for instrumenting application code
- a push gateway for supporting short-lived jobs
- special purpose exporters for services like HAProxy, StatsD, Graphite, etc.
- an alertmanager to handle alerts
- various support tools

Most Prometheus components are written in Go, making them easy to build and deploy as static binaries.

Architecture

Prometheus scrapes metrics from instrumented jobs, either directly or via an intermediary push gateway for short-lived jobs. It stores all scraped samples locally and runs rules over this data to either aggregate and record new time series from existing data or generate alerts. Grafana or other API consumers can be used to visualize the collected data.

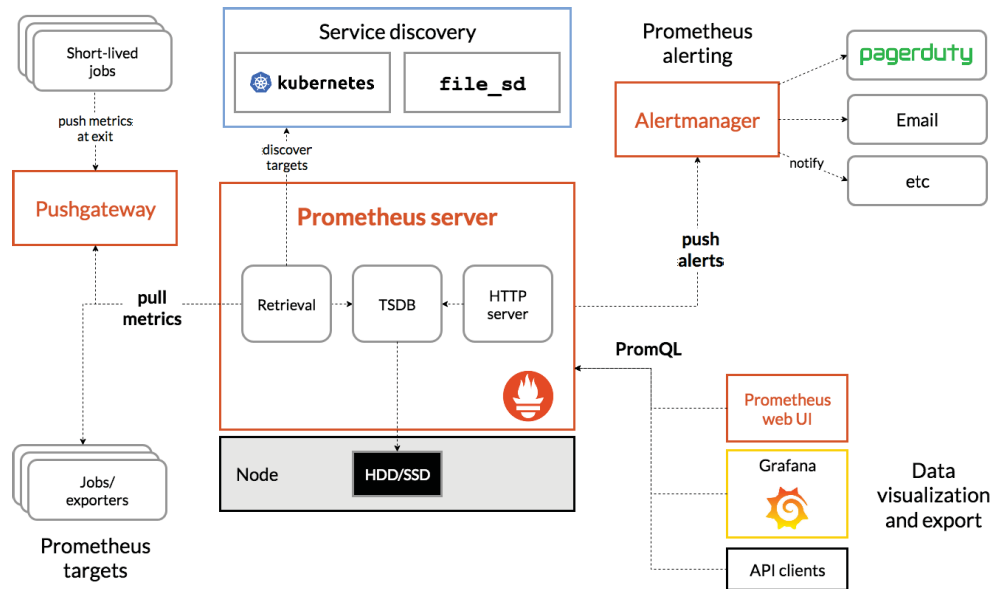


Figure 4.3 Prometheus Architecture

In our approach, we are more interested in the values than the graphs at the time of metrics collection. So, Grafana is not involved to visualise the graphs. Instead PromQL queries are directly used in Prometheus web UI and the port was queried externally to store the values.

4.3 Experimental Setup

As mentioned, the main purpose of this thesis is to identify how Kubernetes and Docker Swarm are affected from scaling the users in video call application and determine the best suited orchestration tool. Cloud services were not considered as a simple One-to-One call and One-to-Many call is to be implemented with Kurento framework on a single node cluster. All the parameters such as number of nodes in cluster of the orchestration tool, number of replicas of containers, resources in the system and duration of experiment other than number of users or calls were considered constant. The load on Kurento server is increased with time by adding number of connections. Each connection adds two users to the server in One-to-One call application and one viewer for One-to-Many call application. With features like autoscaling in cloud services, we proceeded with local implementation.

The initial experiment was setup in local Ubuntu 18.04 desktop with configuration of 8GB RAM. Due to experiment with Minikube, we have created a virtualized environment (VirtualBox) to setup the docker and Kubernetes with kubeadm. The VM was created with 4GB RAM and 20GB disk space at first, but the application is not that smooth. So, a new VM is created with 6GB and installations of docker, kubeadm are done. It is accessed in the local Ubuntu desktop through SSH. This method is effective as we can launch multiple VMs in case we extend the thesis to work with worker nodes and operate the whole setup from local desktop.

Before we start the application, the cluster will be setup either by resetting the configuration and network of Kubernetes or machine joining the Swarm. The Manager node becomes alive and launches the containers in the orchestrated environment, one has the application running on it and the other container has the Kurento Media server. The application and Kurento Media Server were discovered by each other through the services defined in the cluster.

Once these containers are up and running, we can be able to make a call from the browser. Then Prometheus is launched, CPU, Memory, Network Traffic and Packet Loss was collected in each scenario and are measured. Performing the experiment yield results on which statistical analysis is done using both statistical and probability principles. We will establish trade-offs among the tools used in terms of Network Traffic, Packet Loss, memory and CPU Utilization which are the performance metrics considered and features of the respective orchestration tools also will be taken into consideration.

4.3.1 Software Requirements

The software used are.

- Virtual Box 5.2.42
- Virtual machines with Ubuntu 20.04 OS (installed in Virtual Box)
- Ubuntu Host machine with Ubuntu desktop 18.04 LTS
- Docker 20.10.7
- Kubeadm 1.20.4/1.20.10
- Docker Compose 1.17.1
- Kurento media server 6.14.0
- Prometheus

4.3.2 System Requirements

This section describes the minimum hardware and network requirements for the setup.

Hardware Requirements

Plan the hardware for the Virtual machines with

- At least 6GB RAM for Ubuntu desktop VM(8GB would be good)
- At least 20GB of Disk space, or more as per the necessity

Network Requirements

A network adapter is required for the Virtual Box VM's to set up Bridged network between the host and virtual machine.

4.4 Implementation

4.4.1 Kubernetes

The Kubernetes cluster was created using kubeadm, two deployments were created either for One-to-One call or One-to-Many call and for Kurento Media Server. For each of the deployment a service was created. Now we have two pods running who can reach other through services. The ports on which Kurento media server and application running are exposed with services so they can be accessed outside the cluster. All these details are executed through YAML files.

Once application is running, another Prometheus container is launched which scrapes the metrics through Node Exporter. This is also exposed through a service.

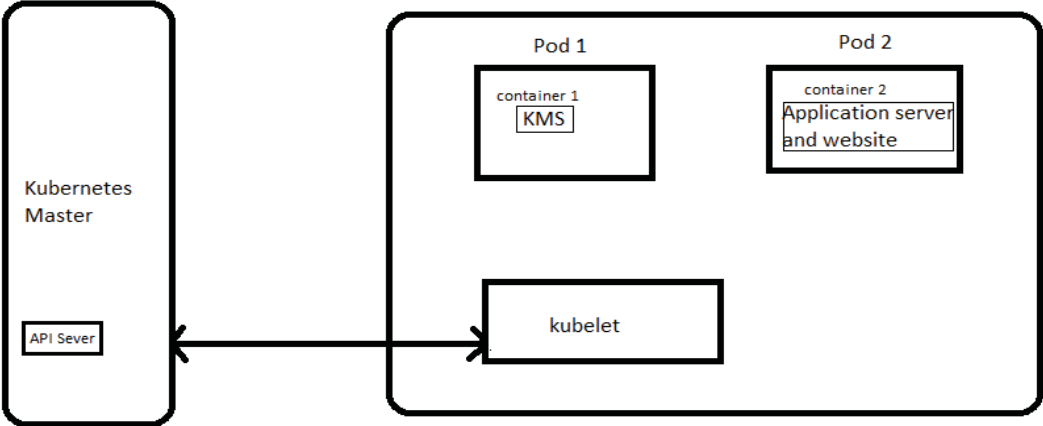


Figure 4.4 Kubernetes node

4.4.2 Docker Swarm

Docker Swarm is already included in the docker for managing clusters. The IP address of docker machine which is considered as the manager is advertised. Then the swarm manger returns a token command which can be executed in other docker machines to join the swarm as workers. Here we limited to the manager node. The containers are launched with stack deployment using the docker-compose file. Here unlike Kubernetes, default network and containers are up with stack deploy command. The default network created makes containers or the tasks to find each other. This way, even though application and Kurento are running in different containers, they work together.

In Docker Swarm, Prometheus is up and running along with the application containers as it is launched in the same docker-compose file.

Docker Swarm

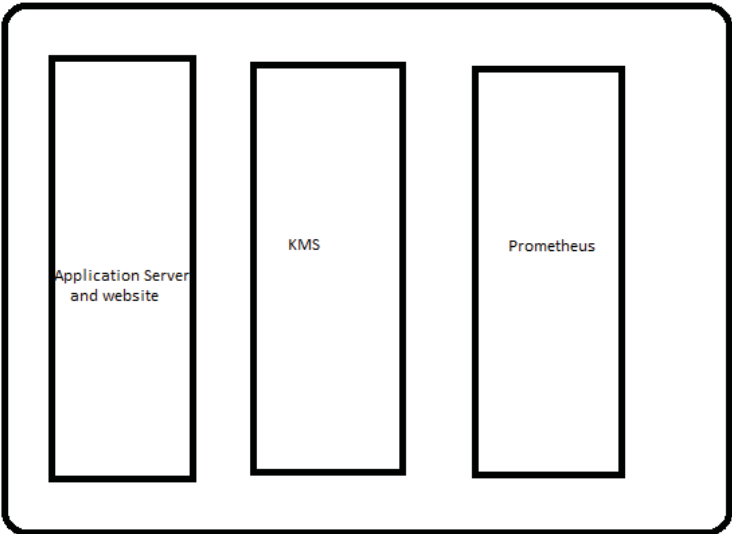


Figure 4.5 Docker Swarm node

4.5 Experimental Scenarios

This section explains the different experiment scenarios used for this thesis work using the setup illustrated in the section 4.3

4.5.1 One-to-One Call Application

In this scenario, the standard One-to-One call is launched as the client-server model in local ubuntu desktop and then in both Kubernetes and Docker Swarm.

First, only one call with two users were present in the call. Different metrics like CPU Utilization, Memory Utilization, Packet Loss, Network Traffic between them is calculated, tabulated and analyzed over the period of thirty minutes. The purpose of this scenario is to establish a baseline for these metrics towards which the overhead of containers can be evaluated.

Next, the number of One-to-One calls were increased at an interval of every five minutes. With each call connection, the same metrics as above were calculated, tabulated and analyzed over the period. This will enable us to compare the orchestration tools and finally we can determine the best suited tool among Kubernetes and Docker Swarm.

4.5.2 One-to-Many Call Application

In this scenario, the One-to-Many call is launched as the client-server model in local ubuntu desktop and then in both Kubernetes and Docker Swarm.

Here two tests were performed, firstly with a presenter and a viewer in the call for thirty minutes. After the first call, another call is made with one presenter and one viewer to begin and scaling viewers for every five minutes(adding each at a time) over the period. In both cases, performance metrics were collected and analyzed.

5 RESULTS AND ANALYSIS

In this chapter, we discuss performance in terms of CPU, Memory Utilization, Packet Loss, Network Traffic for the scenarios presented in section 4.5. The first scenario is the One-to-One call with three use cases and the second is a One-to-Many call. We are interested in these metrics so that load on the Kurento server, network overhead and better service consuming less resources can be estimated by careful observations and analysis.

In a real environment, the Kurento media server will have many requests to process from across the network. For experimental purpose, we only considered the calls within the same network to limit the variables of experiment. Here we are only interested in the overhead of Kurento media server in Kubernetes and Docker Swarm. So, the experiment is performed without orchestration(containerised Kurento Media Server running in the local system) and then in both the mentioned orchestration tools. The experiment is carried out for One-to-One call application(2 users and 14 users) and One-to-Many call application(2 users and 8 users). Each call is conducted for 30 minutes, monitoring the performance metrics resulting 30 average values per minute in every case. Confidence intervals for the mean values of them are also calculated to estimate the variability of the results.

Note that the values used for plotting the graphs in the results are rounded to 2 significant figures.

5.1 CPU Utilization

CPU Utilization tells the amount of work handled by the CPU for processing resources and varies with the amount and type of computing tasks managed. The CPU Utilization is considered to estimate the additional load on CPUs of containerised Kurento servers compared to standard client-server model(only Kurento media server is containerised). For an operating system, the frequency of the clock may vary in the range 100-1000HZ (i.e., 100-1000 ticks/second). The CPU clock is used time the instructions inside the CPU (thus is not available to software), whereas the software clock is used for process scheduling. Processes are allocated a time slice during which they can execute. This time slice is expressed in the number of ticks. At a given time, a CPU (or core) can be either active (using 100% of its capacity) or idle (0%).

With increase in number of streams and connections, the Kurento server utilises more and more CPU. So, it is important to analyse the impact of CPU on the system performance. CPU Utilization is calculated using top command in standard One-to-One and One-to-Many calls. CPU when Kurento server orchestrated in Kubernetes and Docker Swarm is calculated using PromQL queries through Prometheus tool.

In scenario 1, for standard One-to-One call, the percent of CPU Utilization per process running on port 8888 is calculated from top command. CPU Utilization is monitored for every second till 60 seconds to estimate the average CPU Utilization per minute. 30 such average CPU Utilization samples are obtained in percentage. Later, Mean, Standard Deviation, 95% confidence interval with its relative value from mean are also calculated for the obtained samples of average CPU Utilization per minute.

In both Kubernetes and Docker Swarm, the CPU is measured with Prometheus, monitored every 15 seconds and averaged per minute to estimate the utilization per minute. The One-to-One call with 2 users is monitored for thirty minutes and then 7 connections referred as calls with 14 users are monitored in the same procedure.

CPU Utilization per minute	One-to-One call	Mean (%)	Standard Deviation	95% Confidence Interval (%)
Without Orchestration	Two users	7.946222	0.56004	+/- 0.200407641
	Fourteen users	13.515939	4.9457448	+/- 1.769812
Kubernetes	Two users	0.082474915	0.006368902	+/- 0.002279082
	Fourteen users	0.214091979	0.18734926	+/- 0.067042072
Docker Swarm	Two users	7.815679594	0.414535612	+/- 0.148339663
	Fourteen users	14.4671355	3.449708604	+/- 1.23446237

Table 5.1 Average CPU Utilization per minute of Kurento Server for One-to-One call application

Table 5.1 shows the average CPU Utilization while One-to-One call application using Kurento server without orchestration, along with its relative values from the mean. The average CPU Utilization per minute on the server for 2 users(one connection) is approximately 7.9% and the average CPU Utilization per minute for 14 users(seven connections) is approximately 13.5%. From Table 5.1, the average CPU Utilization per minute on the server in Kubernetes for 2 users(one connection) is approximately 0.1% and the average CPU Utilization per minute for 14 users(seven connections) is approximately 0.2%. From Table 5.1, the average CPU Utilization per minute on the server in Docker Swarm for 2 users(one connection) is approximately 7.8% and the average CPU Utilization per minute for 14 users(seven connections) is approximately 14.4%.

For scenario 2, One-to-Many call is also analyzed in the same procedure as above with 2 users to start with and scaled to 8 users. Then the experiments are carried in Kubernetes and Docker Swarm where a presenter and a viewer are monitored for thirty average values and then new call with a Presenter and a viewer along with additional 6 connections referred as calls with total 8 users are monitored for another 30 iterations. The average CPU Utilization in percent(%) is calculated on server and hosts from both Kubernetes and Docker Swarm and tabulated along with the relative values from mean.

Table 5.2 shows the average CPU Utilization while One-to-Many call application using Kurento server without orchestration, along with its relative values from the mean. The average CPU Utilization per minute on the server for 2 users(one connection) is approximately 4.25% and the average CPU Utilization per minute for 8 users(seven connections) is approximately 6.35%. From Table 5.2, the average CPU Utilization per minute on the server in Kubernetes for 2 users(one connection) is approximately 0.1% and the average CPU Utilization per minute for 8 users(seven connections) is approximately 0.19%. From Table 5.2, the average CPU Utilization per minute on the server in Docker Swarm for 2 users(one connection) is approximately 5.68% and the average CPU Utilization per minute for 8 users(seven connections) is approximately 7.03%.

CPU Utilization per minute	One-to-Many call	Mean (%)	Standard Deviation	95% Confidence Interval (%)
Without Orchestration	Two users	4.253111	0.810122	+/- 0.289898312
	Eight users	6.359888	1.314303	+/- 0.470317127
Kubernetes	Two users	0.102357106	0.036973434	+/- 0.013230773
	Eight users	0.187665375	0.086581762	+/- 0.030982885
Docker Swarm	Two users	5.681288339	0.321255681	+/- 0.114959869
	Eight users	7.026439241	1.596696004	+/- 0.571370327

Table 5.2 Average CPU Utilization per minute of Kurento Server for One-to-Many call application

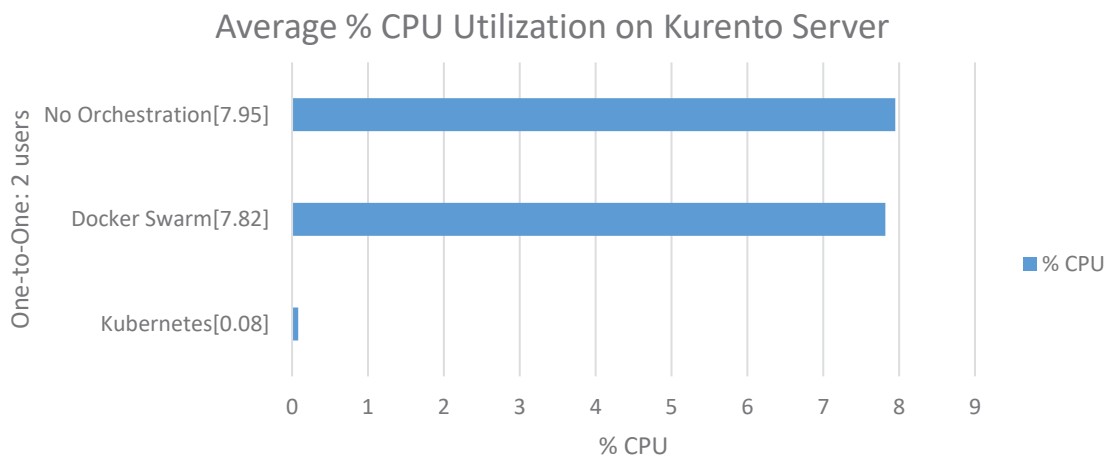


Figure 5.1 Average CPU Utilization on Kurento Server for One-to-One call(2 users)

The Figure 5.1 shows that for scenario 1 with two users utilizes only 7.9% of the CPU but in case of Kubernetes, it dropped to 0.1% approximately. There is not much difference between Docker Swarm and no orchestration use cases, latter with an additional 0.13% approximately. The CPU utilised by Kurento server alone is reduced with orchestration when compared with deploying the stand-alone containers.

The Figure 5.2 shows that for scenario 1 with fourteen users utilizes 13.51% thus an extra load of 5.56% is required when 2 users become fourteen. This utilization is way higher than Kubernetes which is approximately 0.2%. The highest utilization is observed in Docker Swarm with an approximate percent of 14.46%. The load of 0.95% is observed on Docker Swarm with that of no orchestration case. There is almost double the load on Kurento server when 2 users are scaled to 14 users.

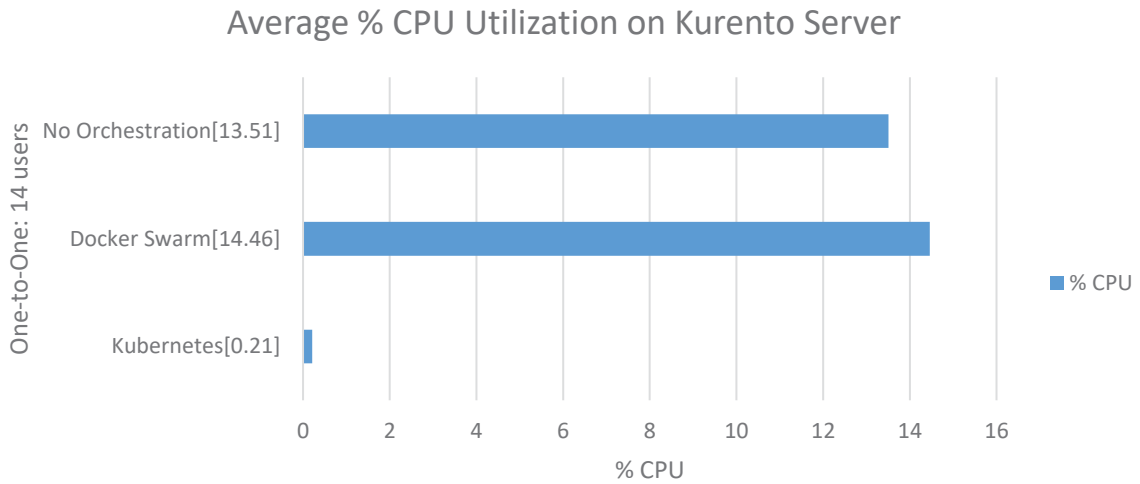


Figure 5.2 Average CPU Utilization on Kurento Server for One-to-One call(14 users)

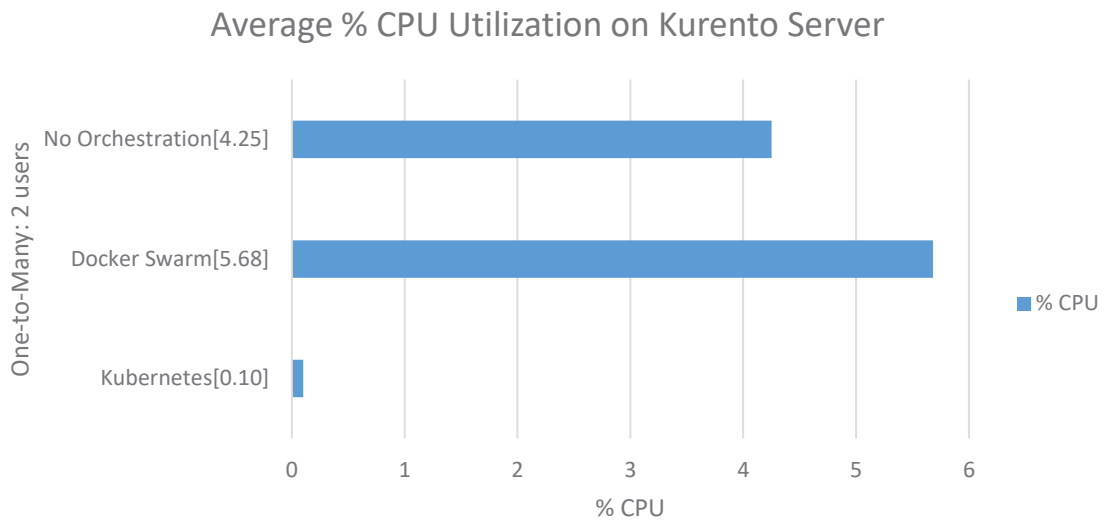


Figure 5.3 Average CPU Utilization on Kurento Server for One-to-Many call(2 users)

The Figure 5.3 shows that for scenario 2 with two users utilizes only 4.3% of the CPU but in case of Kubernetes, it dropped to 0.1% approximately. Kubernetes load for One-to-One call is similar to this One-to-Many two users call. The CPU Utilization is reduced in case of Docker Swarm from previous case, but it topped among the values grouped with approximate value of 5.7%.

The Figure 5.4 shows that for scenario 2 with eight users, Kurento server without orchestration utilizes 6.35% which is almost half of what One-to-One call utilized for fourteen. The same pattern is observed in Docker Swarm with a value of 7.03%. Kubernetes has not much difference in this use case when compared with One-to-One call.

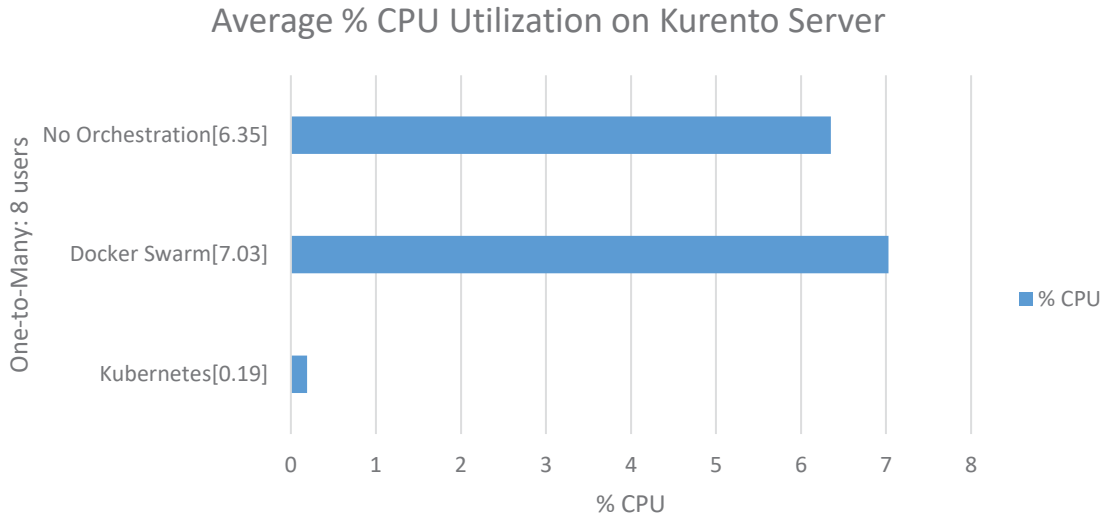


Figure 5.4: Average CPU Utilization on Kurento Server for One-to-Many call(8 users)

Kubernetes has minimum difference of CPU Utilization in terms of different applications. The CPU load only varied with increase in number of users. Docker Swarm saw a raise in load from 2 to 14 users for One-to-One call and One-to-Many call separately but the load on CPU is much less on One-to-Many call. In the same way, Kurento server without orchestration experienced an increase in load with increase in number of users. As mentioned, it also saw a drop in load when experiment on One-to-Many call application is performed. For One-to-One call applications, the order of CPU Utilizations in ascending order is as follows: Kubernetes, Docker Swarm and server without orchestration. Whereas for One-to-Many call, Kubernetes, server without orchestration and then Docker Swarm.

5.2 Memory Utilization

Memory usage represents the proportion of the total amount of random access memory used and is an important parameter related to the performance of a system. The Memory Utilization is considered to estimate the overhead on containerised Kurento server compared to standard client-server model. RAM is the short-term memory on which system relies on for quick access to run all processes fast. A system will slow down if it exceeds the RAM capacity because it will be forced to swap memory pages to disk. From the total memory, the memory in use by the OS is called as used and the memory not in use is called as free. But if applications request memory the OS will free the buffers and cache created from the available memory supporting the system to run faster. The buffers and cache show memory usage for specific purposes, these values are included in the value for used. We are obtaining memory statistics based on top command and Prometheus.

Kurento Media Server needs to process the data continuously, it is important to analyze and estimate the amount of memory required for each connection. Hence impact of adding more and more connections on the memory is also a crucial thing which needs to be considered. A high memory usage on the server may lead to the poor performance of the application. Here in this research, we considered the amount of the memory in RAM consumed during the implementation of the orchestration tools that are considered in this research. In Linux, we have contents of the memory usage associated to a process. We considered top command and Prometheus for measuring the Memory Utilization percentages in given scenarios.

In scenario 1, for standard One-to-One call, the percent of Memory Utilization per process running on port 8888 is calculated from top command. Memory Utilization is monitored for every second till 60

seconds to estimate the average Memory Utilization per minute. 30 such average Memory Utilization samples are obtained in percentage. Later, Mean, Standard Deviation, 95% confidence interval with its relative value from mean are also calculated for the obtained samples of average Memory Utilization per minute.

In both Kubernetes and Docker Swarm, the memory used is measured in bytes with Prometheus, monitored every 15 seconds and averaged per minute to estimate the utilization per minute. Later the obtained values are converted to percentage by dividing with total memory and then multiplying with 100. The One-to-One call with 2 users is monitored for thirty average values and then 7 connections referred as calls with 14 users are monitored in the same procedure.

Memory Utilization per minute	One-to-One call	Mean (%)	Standard Deviation	95% Confidence Interval (%)
Without Orchestration	Two users	0.454889	0.049756	+/- 0.017804949
	Fourteen users	0.5723889	0.1466998	+/- 0.052495849
Kubernetes	Two users	31.43479761	19.65680567	+/- 7.034097572
	Fourteen users	12.6988312	22.58218565	+/- 8.080931352
Docker Swarm	Two users	12.35324726	15.19445886	+/- 5.437267272
	Fourteen users	3.086505411	4.817691193	+/- 1.723988653

Table 5.3 Average Memory Utilization per minute of Kurento Server for One-to-One call application

Table 5.3 shows the average Memory Utilization while One-to-One call application using Kurento server without orchestration, along with its relative values from the mean. The average Memory Utilization per minute on the server for 2 users(one connection) is approximately 0.45% and the average Memory Utilization per minute for 14 users(seven connections) is approximately 0.57%. From Table 5.3, the average Memory Utilization per minute on the server in Kubernetes for 2 users(one connection) is approximately 31.43% and the average Memory Utilization per minute for 14 users(seven connections) is approximately 12.7%. From Table 5.3, the average Memory Utilization per minute on the server in Docker Swarm for 2 users(one connection) is approximately 12.35% and the average CPU Utilization per minute for 14 users(seven connections) is approximately 3.08%.

For scenario 2, One-to-Many call is also analyzed in the same procedure as above with 2 users to start with and scaled to 8 users. Then the experiments are carried in Kubernetes and Docker Swarm where a presenter and a viewer are monitored for thirty iterations and then new call with a Presenter and a viewer along with additional 6 connections referred as calls with total 8 users are monitored for another 30 iterations. The average Memory Utilization in % is calculated on server from both Kubernetes and Docker Swarm and tabulated along with the relative values from mean.

Memory Utilization per minute	One-to-Many call	Mean (%)	Standard Deviation	95% Confidence Interval (%)
Without Orchestration	Two users	0.401778	0.020412	+/- 0.007304387
	Eight users	0.4505	0.051251	+/- 0.018339901
Kubernetes	Two users	4.714091731	0.883349639	+/- 0.316102609
	Eight users	1.789574463	2.17027701	+/- 0.776623654
Docker Swarm	Two users	1.595003569	2.666111442	+/- 0.954055727
	Eight users	3.492656176	3.65094386	+/- 1.306473481

Table 5.4 Average Memory Utilization per minute of Kurento Server for One-to-Many call application

Table 5.4 shows the average Memory Utilization while One-to-Many call application using Kurento server without orchestration, along with its relative values from the mean. The average Memory Utilization per minute on the server for 2 users(one connection) is approximately 0.40% and the average Memory Utilization per minute for 8 users(seven connections) is approximately 0.45%. From Table 5.4, the average Memory Utilization per minute on the server in Kubernetes for 2 users(one connection) is approximately 4.71% and the average Memory Utilization per minute for 8 users(seven connections) is approximately 1.79%. From Table 5.4, the average Memory Utilization per minute on the server in Docker Swarm for 2 users(one connection) is approximately 1.59% and the average Memory Utilization per minute for 8 users(seven connections) is approximately 3.49%.

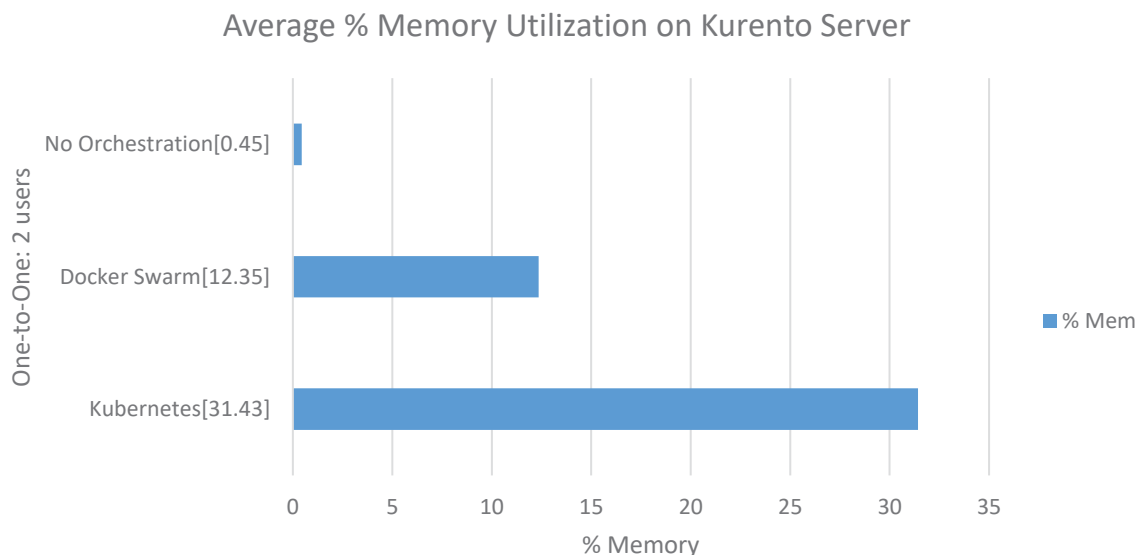


Figure 5.5: Average Memory Utilization on Kurento Server for One-to-One call(2 users)

The Figure 5.5 shows that for scenario 1 with two users, Kurento server without orchestration utilizes only 0.5% of the memory but in case of Kubernetes, it is high with 31.5% consumption. Docker Swarm averaged between Kubernetes and server without orchestration with a value of 12.35%. Kubernetes Memory Utilization is approximately 19% higher than Docker Swarm and 31% higher than no orchestration case.

The Figure 5.6 shows that for scenario 1 with eight users, non-orchestrated server utilizes 0.12% higher than 2 users but lower in the group with a value of 0.57%. Kubernetes has highest percent with 12.7%. Docker Swarm has less consumption compared to Kubernetes in terms of Memory Utilization around 9% approximately with 3%. But in comparison overall values of consumption for 14 users is less than 2 users.

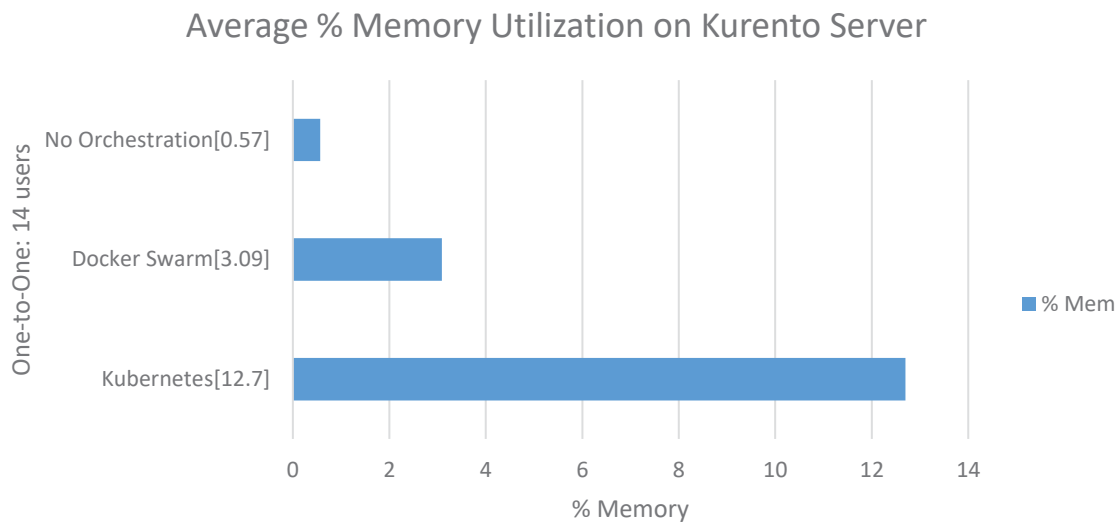


Figure 5.6 Average Memory Utilization on Kurento Server for One-to-One call(14 users)

The Figure 5.7 shows that for scenario 2 with two users, Kurento server without orchestration utilizes only 0.4% of the memory but in case of Kubernetes, it is high with 4.7% consumption. Docker Swarm averaged between Kubernetes and server without orchestration with a value of 1.6%. Kubernetes Memory Utilization is approximately 3% higher than Docker Swarm and 4% higher than no orchestration case.

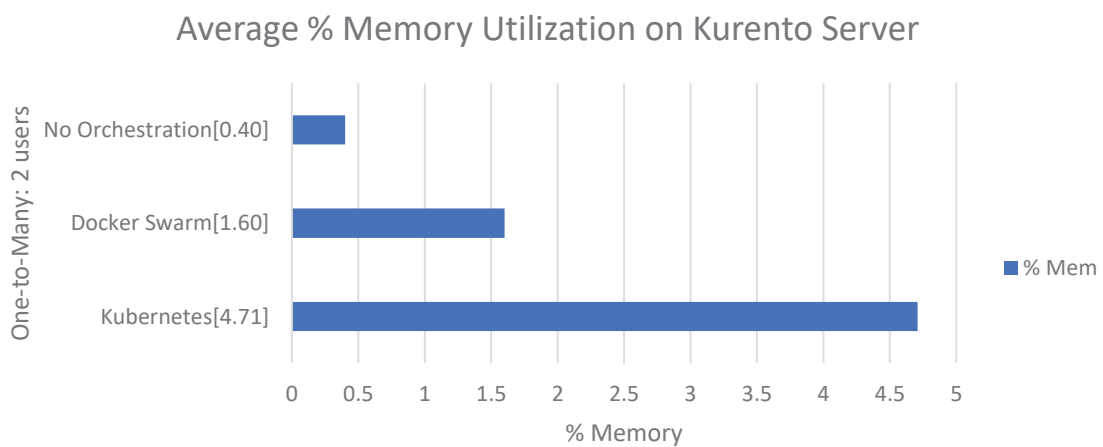


Figure 5.7 Average Memory Utilization on Kurento Server for One-to-Many call(2 users)

The Figure 5.8 shows that for scenario 2 with eight users, non-orchestrated server utilizes 0.45%. Docker Swarm has highest consumption with 3.5% twice the consumption of Kubernetes which averaged only 3.5% in this scenario. But in comparison overall values of consumption for 14 users is less than 2 users for both Kubernetes and Docker Swarm.

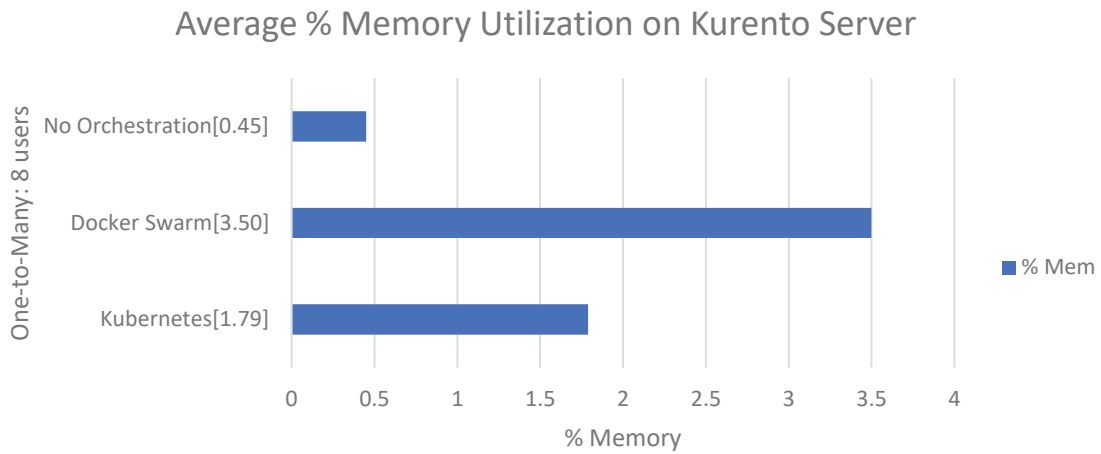


Figure 5.8 Average Memory Utilization on Kurento Server for One-to-Many call(8 users)

The Memory Utilization in orchestrated server is higher than the non-orchestrated server. Kubernetes showed higher consumption than compared to Docker Swarm within the orchestration environment. Kubernetes showed a consistent increase from One-to-One call with 2 users and 14 users. Docker Swarm saw a raise in consumption in One-to-Many call with 8 users. In the same way, Kurento server without orchestration experienced an increase in load with increase in number of users.

5.3 Network Traffic

Network Traffic is the sequence of encapsulated network packets flowing across a network from a source to destination. The Network Traffic is yet another metric we used to quantify the overhead due to orchestration. The overhead in terms of traffic can be estimated by measuring the amount (Kb/s) and type (UDP/TCP) of traffic on a network. The traffic is usually measured in bits or bytes. However, we expressed the data rate of traffic in Kilobits (Kb/s).

WebRTC is to actually be able to serve video in a secure, reliable and effective manner, and this must include being able to respond appropriately when the network connectivity of the viewers is flaky and suffers from all sorts of real-world issues, such as congestion, packet loss, and other types of misbehaviors of the network. Kurento Media Server offers a comprehensive solution that covers all of the described points. Acquiring a video stream from a variety of sources, together with the optional transcoding of the media, is performed by the PlayerEndpoint. Then everything related to the WebRTC communications is handled by the aptly named WebRtcEndpoint. So, estimating the Network Traffic in each of the experimental scenario is important to monitor the performance of servers and give a better experience for the end user.

We can easily capture the Network Traffic using Wireshark for port 8888 running Kurento server without orchestration by telling it which interface to capture traffic on or providing the appropriate capture filters to capture only the necessary traffic, in addition duration of the capture can also be specified. This method is followed for both the scenarios. The total packet traffic statistics can be viewed in Wireshark from the statistics of the pcap file. Here, we can see the measurement statistics of captured interface, capture filter, packets captured, average packets per second, average packet size, bytes captured, average bytes per second, average bits per second and few other details. Wireshark also

supports terminal oriented version of it called t-shark designed to capture traffic when an interactive user interface is not available or necessary and allows to analyze the captured file by displaying from Wireshark.

Traffic on port 8888 of the server is captured by Wireshark from the host system for duration of 60 seconds to estimate the average Kilobits per second. 30 such Average Kb/s samples are obtained. Later, Mean, Standard Deviation, 95% confidence interval with its relative value from mean are also calculated for the obtained samples of average Kb/s.

Network Traffic per second	One-to-One call	Mean (Kb/s)	Standard Deviation	95% Confidence Interval (Kb/s)
Without Orchestration	Two users	293.7	56.08193	+/- 20.06866055
	Fourteen users	712.4603	203.7262	+/- 72.902474
Kubernetes	Two users	0.008701794	0.00715846	+/- 0.002561622
	Fourteen users	0.517640768	1.347383909	+/- 0.482155139
Docker Swarm	Two users	295.0498207	50.35272789	+/- 18.01849227
	Fourteen users	639.9850089	193.0311291	+/- 69.07530242

Table 5.5 Average Network Traffic per second of Kurento Server for One-to-One call application

Table 5.5 shows the average Network Traffic per second while One-to-One call application using Kurento server without orchestration, along with its relative values from the mean. The average Network Traffic per second on the server for 2 users(one connection) is approximately 293.7 Kb/s and the average Network Traffic per second per minute for 14 users(seven connections) is approximately 712.46 Kb/s. From Table 5.5, the average Network Traffic per second per minute on the server in Kubernetes for 2 users(one connection) is approximately 0.008 Kb/s and the average Network Traffic per second for 14 users(seven connections) is approximately 0.517 Kb/s. From Table 5.5, the average Network Traffic per second on the server in Docker Swarm for 2 users(one connection) is approximately 295.04 Kb/s and the average Network Traffic per second for 14 users(seven connections) is approximately 639.96 Kb/s.

For scenario 1 and 2, in Kubernetes and Docker Swarm, the transmitted and received Network Traffic of Kurento media server is captured by Prometheus separately and then average of total traffic per minute is recorded. 30 such average values are obtained. Network Traffic is converted from bytes to Kilobytes per second.

Network Traffic per second	One-to-Many call	Mean (Kb/s)	Standard Deviation	95% Confidence Interval (Kb/s)
Without Orchestration	Two users	148.6333	18.45682	+/- 6.604686454
	Eight users	343.5	156.2981	+/- 55.93054549
Kubernetes	Two users	0.047146025	0.237746737	+/- 0.085076577
	Eight users	0.19392788	0.438252347	+/- 0.156826588
Docker Swarm	Two users	153.6182747	17.78069162	+/- 6.36273878
	Eight users	340.3452479	143.8619414	+/- 51.48033458

Table 5.6 Average Network Traffic per second of Kurento Server for One-to-Many call application

Table 5.6 shows the average Network Traffic per second while One-to-Many call application using Kurento server without orchestration, along with its relative values from the mean. The average Network Traffic per second on the server for 2 users(one connection) is approximately 148.63 Kb/s and the average Network Traffic per second per minute for 8 users(seven connections) is approximately 343.5 Kb/s. From Table 5.6, the average Network Traffic per second per minute on the server in Kubernetes for 2 users(one connection) is approximately 0.04 Kb/s and the average Network Traffic per second for 8 users(seven connections) is approximately 0.19 Kb/s. From Table 5.6, the average Network Traffic per second on the server in Docker Swarm for 2 users(one connection) is approximately 153.61 Kb/s and the average Network Traffic per second for 8 users(seven connections) is approximately 340.34 Kb/s

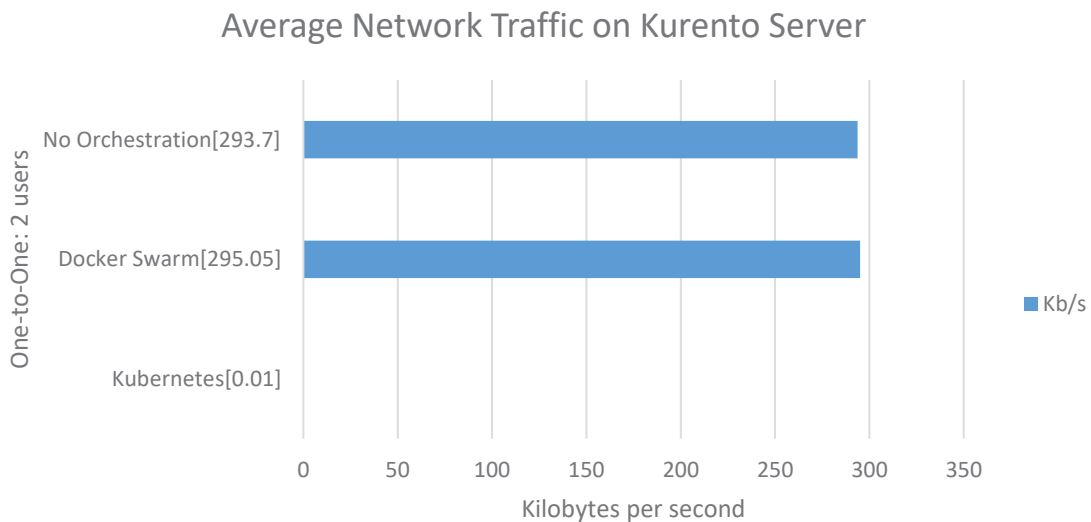


Figure 5.9 Average Network Traffic on Kurento Server for One-to-One call(2 users)

The Figure 5.9 represents scenario 1 of One-to-One call with two users, Kurento server without orchestration has an average traffic of 293.7 Kbps. Docker Swarm has high traffic than Kubernetes and server without orchestration with a value of 295.05 Kbps. This traffic is slightly higher than the non-orchestrated server with a difference of 1.35 Kbps. Kubernetes has very low traffic of 0.01 Kbps as it filters out the requests as per the deployment and remain idle when there is no addition of a new connection or a user.

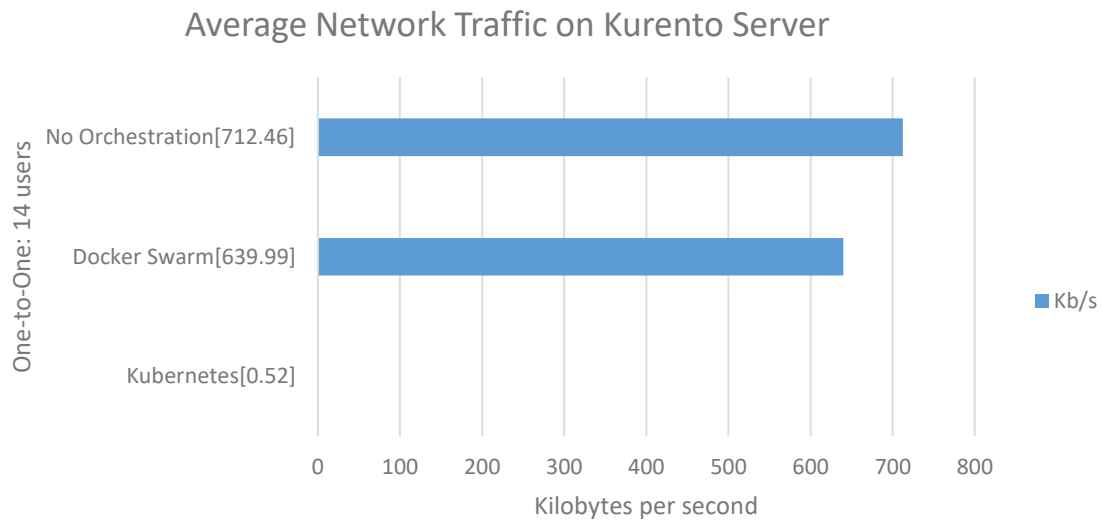


Figure 5.10 Average Network Traffic on Kurento Server for One-to-One call(14 users)

The Figure 5.10 represents scenario 1 of One-to-One call with fourteen users, Kurento server without orchestration has an average traffic of 712.46 Kbps. Docker Swarm averaged between Kubernetes and server without orchestration with a value of 639.99 Kbps. Kubernetes has average traffic of 0.52 Kbps higher than 2 users traffic but less than Docker Swarm and server without orchestration.

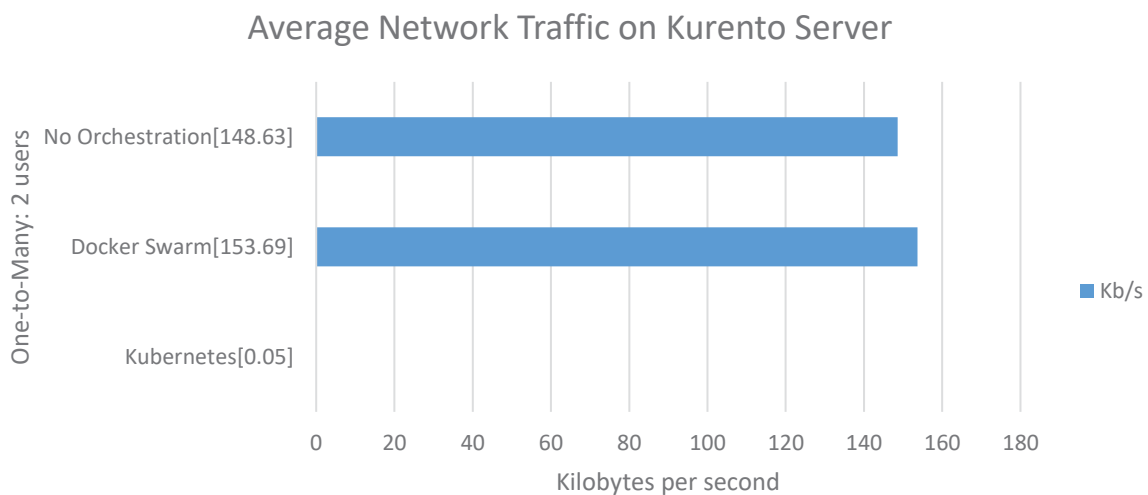


Figure 5.11 Average Network Traffic on Kurento Server for One-to-Many call(2 users)

The Figure 5.11 represents scenario 2 of One-to-Many call with two users, Kurento server without orchestration has an average traffic of 148.63 Kbps. Docker Swarm traffic is highest with a value of 153.69 Kbps. Kubernetes has lowest average traffic of 0.05 Kbps.

The Figure 5.12 represents scenario 1 of One-to-Many call with eight users, Kurento server without orchestration has an average traffic of 343.50 Kbps. Docker Swarm averaged between Kubernetes and server without orchestration with a value of 340.35 Kbps. Kubernetes has average traffic of 0.19 Kbps less than Docker Swarm and server without orchestration.

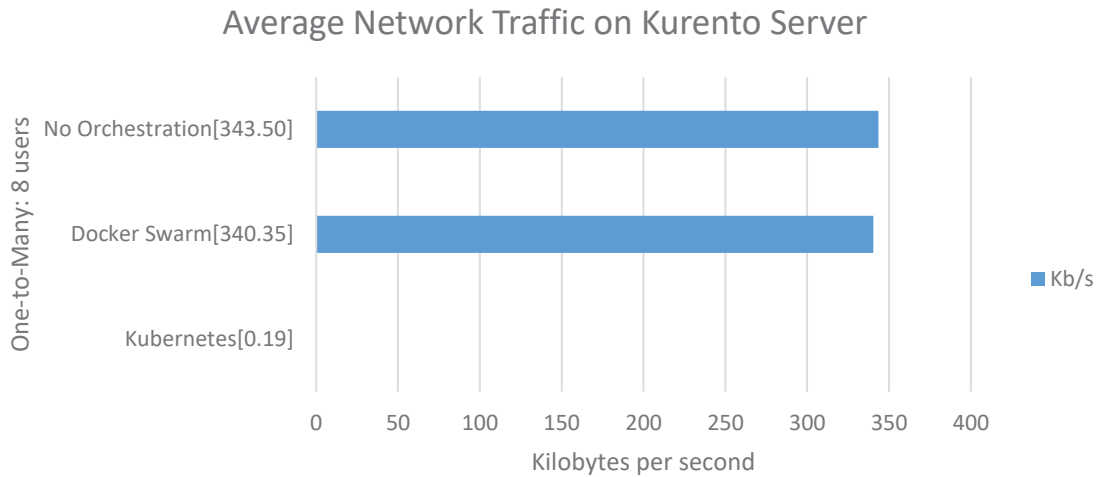


Figure 5.12 Average Network Traffic on Kurento Server for One-to-Many call(8 users)

Network Traffic on non-orchestrated Kurento server is high in 8 users and 14 users case. Docker Swarm has high traffic in 2 users scenario. Kubernetes has the least traffic among them. Kurento server in Kubernetes only processes the requests during connections and will be alive throughout the call. So, Network Traffic is high at the time of connection and settles down gradually.

5.4 Packet Loss

Packet Loss occurs when packets of data do not reach their intended destination usually because of network congestion. A test to measure Packet Loss will determine a percentage of loss. A high percentage indicates you are moving more information than the network can support. This causes packets to be dropped. Packet Loss is experienced when a video becomes choppy or glitchy since the information isn't reaching its intended destination. Some common issues with Packet Loss include:

- WebRTC streams (audio, video, or data) can be lost, and experience varying amounts of network delay.
- Video image seems fine, but playback suffers from a lot of stuttering (i.e. it is not smooth, constantly “jumps” around).
- Video playback is smooth (no colour issues, no macroblocks, no excessive stuttering), but the perceived quality of the details is very poor.
- Video contains green or pink patches in some areas.
- Video contains huge blocks (aka. “macroblocks”)

So, to deliver a seamless video experience to the user in real time, measuring the Packet Loss in all experimental scenarios helps us to conclude the better performing orchestration tool.

Packet Loss per minute	One-to-One call	Mean (%)	Standard Deviation	95% Confidence Interval (%)
Without Orchestration	Two users	0	0	+/- 0
	Fourteen users	0	0	+/- 0
Kubernetes	Two users	0	0	+/- 0
	Fourteen users	0	0	+/- 0
Docker Swarm	Two users	0	0	+/- 0
	Fourteen users	0	0	+/- 0

Table 5.7 Average Packet Loss per minute of Kurento Server for One-to-One call application

Table 5.7 shows the average Packet Loss while One-to-One call application using Kurento server without orchestration, along with its relative values from the mean. The average Packet Loss per minute on the server for 2 users(one connection) is 0 and the average Packet Loss per minute for 14 users(seven connections) is 0. From Table 5.7, the average Packet Loss per minute on the server in Kubernetes for 2 users(one connection) is approximately 0 and the average Packet Loss per minute for 14 users(seven connections) is approximately 0. From Table 5.7, the average Packet Loss per minute on the server in Docker Swarm for 2 users(one connection) is 0 and the average Packet Loss per minute for 14 users(seven connections) is 0.

Packet Loss per minute	One-to-Many call	Mean (%)	Standard Deviation	95% Confidence Interval (%)
Without Orchestration	Two users	0	0	+/- 0
	Eight users	0	0	+/- 0
Kubernetes	Two users	0	0	+/- 0
	Eight users	0	0	+/- 0
Docker Swarm	Two users	0	0	+/- 0
	Eight users	0	0	+/- 0

Table 5.8 Average Packet Loss per minute of Kurento Server for One-to-Many call application

Table 5.8 shows the average Packet Loss while One-to-One call application using Kurento server without orchestration, along with its relative values from the mean. The average Packet Loss per minute

on the server for 2 users(one connection) is 0 and the average Packet Loss per minute for 8 users(seven connections) is 0. From Table 5.8, the average Packet Loss per minute on the server in Kubernetes for 2 users(one connection) is approximately 0 and the average Packet Loss per minute for 8 users(seven connections) is approximately 0. From Table 5.8, the average Packet Loss per minute on the server in Docker Swarm for 2 users(one connection) is 0 and the average Packet Loss per minute for 8 users(seven connections) is 0.

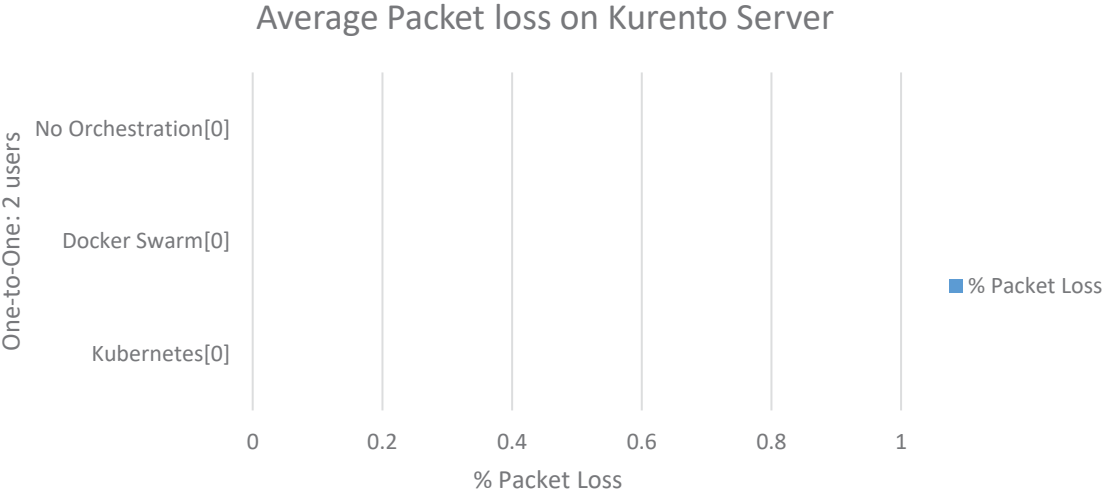


Figure 5.13 Average Packet Loss on Kurento Server for One-to-One call(2 users)

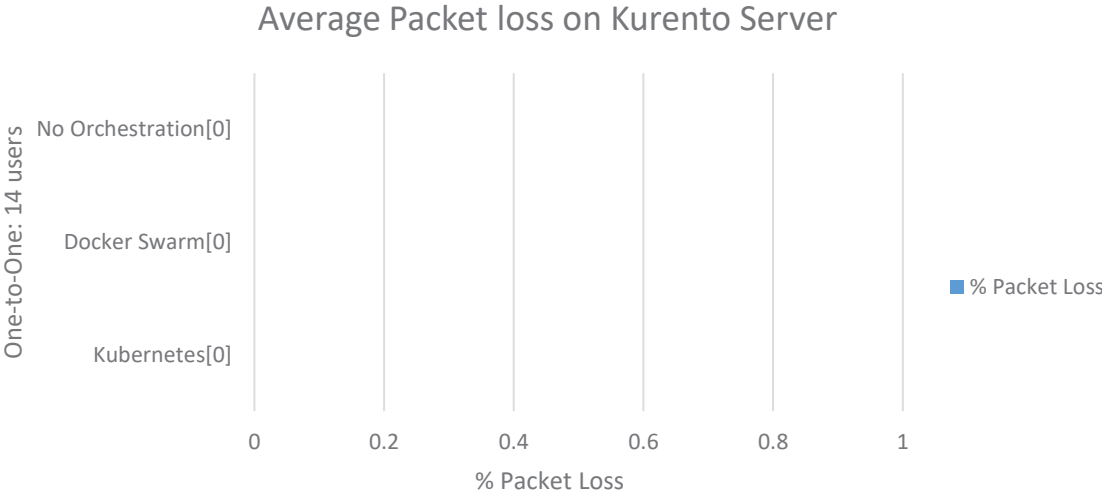


Figure 5.14 Average Packet Loss on Kurento Server for One-to-One call(14 users)

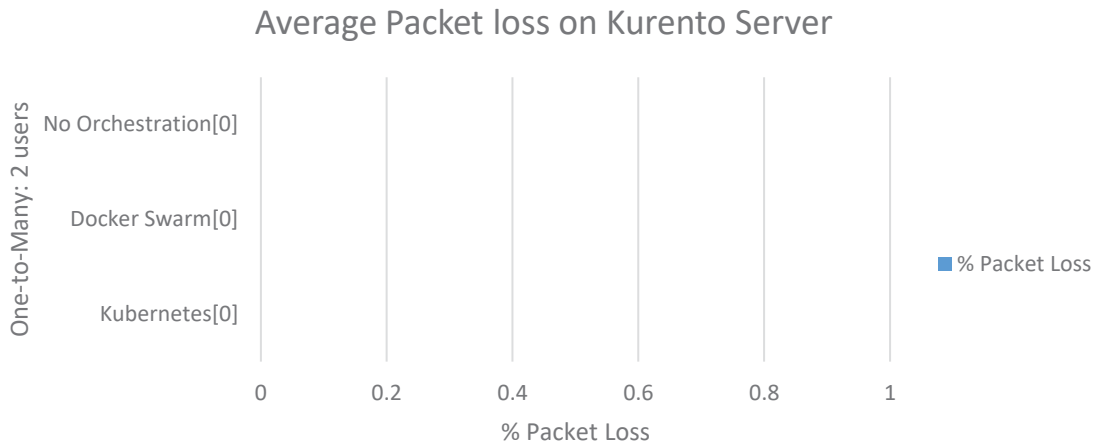


Figure 5.15 Average Packet Loss on Kurento Server for One-to-Many call(2 users)

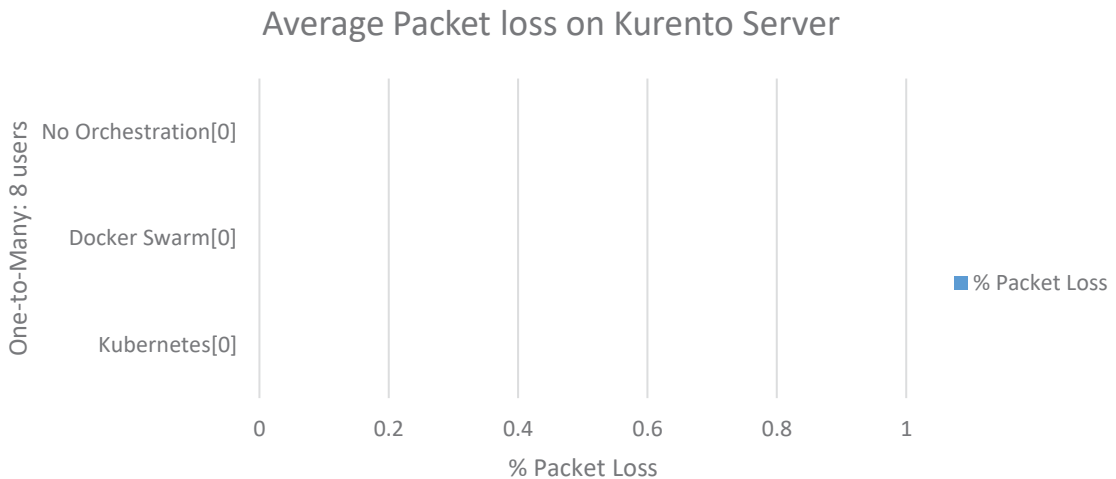


Figure 5.16 Average Packet Loss on Kurento Server for One-to-Many call(8 users)

Packet Loss for all the experimental scenarios resulted in 0%. This shows that Kurento server did not drop any packets while communication and delivered a quality video call. WebRTC has a congestion control algorithm which manages and adapts to the network behaviors in real world. This algorithm controls the congestion in network when there is more or equal to 10% Packet Loss. As experiments are conducted within the same network, Packet Loss might be resulted in 0%. But in real life, Packet Loss can be experienced by the servers. This metric could not help us to determine better orchestration tool, but we learned that the experiment did not experience any loss throughout.

5.5 Analysis Using ANOVA

Analysis of Variance(ANOVA) is an analysis tool used in statistics that splits observed aggregate variability inside a dataset into schematic and random factors. A one-way ANOVA evaluates the impact of a sole factor on a sole response variable. The one-way ANOVA is used to determine whether there are any statistically significant differences between the means of three or more independent (unrelated) groups. Here one way ANOVA test is used for 3 groups of data(No orchestration, Kubernetes and Docker Swarm) to prove that metrics obtained are not similar(at least one).

Below steps are followed to perform one-way ANOVA test in Excel:

1. On the Data tab, in the Analysis group, click Data Analysis.
2. Select Anova: Single Factor and click OK.
3. Click in the Input Range box and select the range of three datasets.
4. Click in the Output Range box and select any cell where we wish to see the output and click OK.

Before analyzing the ANOVA results, we will lay down a hypothesis corresponding to each event, in consideration. They are called the null hypotheses(H_0).

H_{0C} = Mean CPU Utilization of No orchestration, Kubernetes and Docker Swarm is indistinguishable from each other.

H_{0M} = Mean Memory Utilization of No orchestration, Kubernetes and Docker Swarm is indistinguishable from each other.

H_{0N} = Mean Network Traffic of No orchestration, Kubernetes and Docker Swarm is indistinguishable from each other.

Alternate Hypothesis would be: At least one mean(No orchestration, Kubernetes and Docker Swarm) is significantly different from the others.

There are two ways of establishing whether it is possible to reject the null hypotheses. The first way is using the F-distribution table for 95% C.I. The degrees of freedom corresponding to the numerator of the F-score indicate which column in the table, while the degrees of freedom corresponding to the denominator of the F-Score indicate the row. The value in the cell in the table pointed to by the row and the column is the Critical Frequency. Then, the F-Scores are compared against the Critical Frequency.

F-Score>Critical Frequency: Reject the Null Hypothesis(H_0)

F-Score<Critical Frequency: Failed to reject the Null Hypothesis(H_0)

The second way is by looking at the Significance level which indicates the probability of an event. The way that this is interpreted is as follows:

$P < 0.05$: Reject the Null Hypothesis(H_0)

$p > 0.05$: Failed to reject the Null Hypothesis(H_0)

Below are the results obtained from ANOVA test.

One-to-One call	No. Of Users	F value	P-value	F Critical value
CPU Utilization	Two users	3758.551797	3.49541E-85	3.101295757
	Fourteen users	157.0200148	1.35027E-29	3.101295757
Memory Utilization	Two users	35.61112632	5.02283E-12	3.101295757
	Fourteen users	6.914130333	0.001634024	3.101295757

Network Traffic	Two users	457.6266736	6.70758E-47	3.101295757
	Fourteen users	175.3966835	2.97768E-31	3.101295757

Table 5.9 One Way ANOVA test results for One-to-One call

For One-to-One call, from above table 5.9, we observe F-Value is greater than F Critical value in all the scenarios. So, we have enough evidence to reject the Null Hypothesis.

One-to-Many call	No. Of Users	F value	P-value	F Critical value
CPU Utilization	Two users	993.4607824	1.22637E-60	3.101295757
	Eight users	298.6798621	1.08197E-39	3.101295757
Memory Utilization	Two users	56.56425593	1.82872E-16	3.101295757
	Eight users	11.59646587	3.43084E-05	3.101295757
Network Traffic	Two users	1043.294233	1.59175E-61	3.101295757
	Eight users	77.64072	4.48E-20	3.101296

Table 5.10 One Way ANOVA test results for One-to-Many call

For One-to-Many call, from above table 5.10, we observe F-Value is greater than F Critical value in all the scenarios. So, we have enough evidence to reject the Null Hypothesis.

From above analysis and conclusions, we can say that at least any one of the three datasets(No orchestration, Kubernetes and Docker Swarm) perform different from the other but cannot determine which groups(at least one) are significantly different from each another. To investigate further, we perform the post-hoc analysis of the group means by comparing with Standard error. Standard error can be stated as:

$$SE_differences = \sqrt{MS_w / n + MS_w / n}$$

where MS_w is mean square within groups, n is number of samples(30).

As per [59], a group is significantly different when mean difference is more times the standard error. We can observe when mean of Kubernetes is involved, mean difference is much more than the standard error. So, from above, we can strongly infer that mean of Kubernetes is significantly different from the mean of container with no orchestration and Docker Swarm. All the test results can be referred from section 9.2.

6 DISCUSSION

In this section, we mention the aim and objectives of the thesis and describe how they are attained by experiment and analyze how to perform orchestration of Kurento server and calculate its overhead. Research questions were answered emphasising the analysis in previous section.

6.1 Attaining Aim of Thesis

The aim of this work is to investigate the performance of WebRTC based Kurento server in both Kubernetes and Docker Swarm.

To attain the aim, first virtualized environment is setup and docker engine is installed in it. Two kinds of applications were considered for investigation. First is One-to-One video call application where each user can connect with only one another user. This type of application is considered as it is so common in the real world communication. Another being One-to-Many application, where one user presents or streams the data and other users just view it.

Kurento application has three parts, first is the client the user accesses (usually a website). Second is an application server to communicate with KMS (we used NodeJs based server) and the third is Kurento media server itself. We are investigating only the Kurento Media Server here. The user access website and application server were packed into a container through Dockerfile and Kurento media server is launched in a separate container. In Kubernetes and Docker Swarm, these containers find each other through service discovery. Once these containers are deployed, can be accessed through browser and Kurento can be seen running through docker logs.

This experimental setup is investigated using Prometheus which is launched as a separate container in both the setups. Prometheus scrapes the Kubernetes at different levels through Node exporter and cAdvisor. Same way, Docker Swarm is also monitored by Prometheus. CPU, Memory, Network Traffic and Packet Loss are calculated which are averaged per minute. 30 such values are considered and investigated.

So, in this procedure we obtain the objectives by implementing WebRTC based Kurento server and application in Kubernetes and Docker Swarm. Performance is measured using Prometheus in terms of CPU Utilization, Memory Utilization, Network Traffic and Packet Loss. Based on the performance of these, we determined the more suitable recommendation for Web based Real Time Communications.

6.2 Discussing Research Questions

Q1) How does the performance of Kubernetes and Docker Swarm container technologies get affected when application is scaled?

The application used for experiments is scaled in terms of number of users for constant time interval. Two kinds of applications are taken into consideration, One-to-One call and One-to-Many call application.

When One-to-One application is scaled from 2 users to 14 users (7 connection) over 30 minutes with an interval of 5 minutes for each connection (2 users), I observed whenever there is a new connection, there is a spike in CPU Utilization and Memory Utilization. As we are considering the average values per minute, these values are normalized. When we discuss for each performance metric, there is a significant increase in values when application is scaled from 2 to 14 users. CPU Utilization in Kubernetes and Docker Swarm is doubled as represented in Table 5.1. When it comes to memory, I

observed a depletion in both orchestration tools as represented in Table 5.3. Network Traffic is almost doubled in Docker Swarm whereas Kubernetes showed an increase of 50 percent as represented in Table 5.5.

When One-to-Many application is scaled from 2 to 8 users (7 connections) same as scaling One-to-One application, there is a significant spike in all the metrics. With respect to CPU Utilization when compared between 2 users and 8 users, there is an approximate increase of 40 percent in Docker Swarm and doubled for Kubernetes. In case of Memory Utilization, Docker Swarm showed an increase of 100 percent. Kubernetes Memory Utilization decreased. Coming to Network Traffic, in Docker Swarm traffic increased from 153Kbps to 340.35Kbps. Kubernetes Network Traffic increased from 0.05Kbps to 0.19Kbps. These above values can be referred from tables 5.2,5.4 and 5.6 for CPU Utilization, Memory Utilization and Network Traffic respectively.

When application is scaled, the percent of resource utilization increase in Kubernetes is higher when compared to Docker Swarm, but overall resource utilization of Kubernetes is much lower than that of Docker Swarm.

Q2) Is there any significant performance overhead between Kubernetes and Docker Swarm in terms of CPU Utilization, Memory Utilization and Network Traffic?

Orchestration tools manage the resources of containers efficiently than the stand-alone containers. This thesis focuses only on the Kurento server performance overhead rather than application overhead or the resources utilized for launching the master nodes of orchestration tools.

According to data, Kubernetes leaves higher performance overhead[54] when compared to Docker Swarm for its extensive features and ability to handle complex distributed systems. As our main focus is on Kurento server, due to service discovery and load balancing, self-healing and robust functionality features of Kubernetes, the overhead on server is reduced.

From the experimentation and analysis of obtained results, there is a significant overhead in terms of Memory Utilization in comparison with CPU and Network Traffic. When CPU Utilization is considered, the performance overhead in Docker Swarm Vs stand-alone Kurento server container (0.07 for One-to-One 14 users, 0.33 One-to-Many 2 users) is higher than Kubernetes Vs stand-alone Kurento server(-0.98 for One-to-One 14 users, -0.97 One-to-Many 2 users). In case of Memory Utilization, the performance overhead in Kubernetes(68.11 One-to-One 2 users,21.19 for One-to-One 14 users, 10.73 for One-to-Many 2 users) is higher than Docker Swarm (26.2 One-to-One 2 users, 4.39 for One-to-One 14 users, 2.97 for One-to-Many 2 users) except for One-to-Many with 8 users (6.75 for Docker Swarm and 2.97 for Kubernetes). While comparing the Network Traffic overhead, Docker Swarm (0.004 One-to-One 2 users, -0.101 for One-to-One 14 users, 0.03 for One-to-Many 2 users, 0.009 for One-to-Many 8 users) is higher than Kubernetes (-1 One-to-One 2 users, -0.992 for One-to-One 14 users, -148.58 for One-to-Many 2 users, -0.994 for One-to-Many 8 users).

Between Kubernetes and Docker Swarm, later showed the high-performance overhead for CPU and Network Traffic. Memory Utilization overhead is significant in Kubernetes when compared with Docker Swarm.

Q3) Which of the above-mentioned container orchestration technologies is best suited for Web based Real-Time Communication based on previous analysis?

Kubernetes and Docker Swarm are the container orchestration technologies used in this thesis to analyze Kurento server which uses WebRTC. From the experiments performed, it is observed that Kubernetes is better suited for Web based Real-Time Communication when compared with Docker Swarm. We have drawn this conclusion by analyzing CPU Utilization, Memory Utilization and Network Traffic.

Packet Loss is also analyzed but no conclusion can be drawn from the results as mentioned in section 5.4.

During the experiment with One-to-One application, the CPU Utilization in Kubernetes is 0.08 percent whereas in Docker Swarm it is 7.82 percent for 2 users. Similarly for 14 users, the values are 0.21 percent and 14.46 percent respectively. In terms of Memory, Kubernetes has much higher utilization over Docker Swarm with values 31.43 percent and 12.35 percent respectively for 2 users. Whereas for 14 users, the values are 12.7 percent and 3.09 percent for Kubernetes and Docker Swarm respectively. When Network Traffic is taken into consideration for 2 users, Kubernetes has average traffic of 0.01Kbps and Docker Swarm has average traffic of 295.05Kbps. Similarly for 14 users, the values are 0.52Kbps and 639.99Kbps respectively. So, for One-to-One application, Kubernetes offers better performance in terms of CPU and Network Traffic over Docker Swarm and vice versa in case of Memory Utilization

During the experiment with One-to-Many application, the CPU Utilization in Kubernetes is 0.10 percent whereas in Docker Swarm it is 5.68 percent for 2 users. Similarly for 14 users, the values are 0.19 percent and 7.03 percent respectively. In terms of Memory, Kubernetes has higher utilization over Docker Swarm with values 4.71 percent and 1.60 percent respectively for 2 users. Whereas for 14 users, the values are 1.79 percent and 3.50 percent for Kubernetes and Docker Swarm respectively. Here, Kubernetes has higher value than Docker Swarm for 2 users but when application is gradually scaled, Docker Swarm utilized more memory than Kubernetes. When Network Traffic is taken into consideration for 2 users, Kubernetes has average traffic of 0.05Kbps and Docker Swarm has average traffic of 153.69Kbps. Similarly for 14 users, the values are 0.19Kbps and 340.35Kbps respectively. So, for One-to-Many application, Kubernetes offers better performance in terms of CPU, Network Traffic, Memory Utilization (8 users) over Docker Swarm and vice versa in case of Memory Utilization for 2 users.

By analyzing above values, we can understand that One-to-One application utilizes more resources than One-to-Many. This is due to additional streams generated on One-to-One application. In simpler terms, in One-to-One application, video streams are exchanged between both the users whereas in One-to-Many application, Video streaming is done by the presenter exclusively.

Even though we observed difference in values, we cannot confirm if the differences in these values are due to normal statistical fluctuations (non-significant), or unlikely to have occurred by chance (significant). To prove these occurred by chance, I have performed ANOVA test. From ANOVA test it is clear that at least any one of the three datasets (No orchestration, Kubernetes and Docker Swarm) perform different from the other but cannot determine which groups (at least one) are significantly different from each another. To determine which dataset differs from the others, I have performed post-hoc difference of standard errors test and compared Mean differences with standard error. This is explained in section 5.5 and 9.2. From these tests we can conclude Kubernetes means are significantly different (performs better in our use case) from others and this has occurred by chance (significant).

7 CONCLUSION AND FUTURE WORK

7.1 Conclusion

In this thesis, we have analyzed WebRTC based Kurento media server using Kubernetes and Docker Swarm in terms of CPU Utilization, Memory Utilization, Network Traffic and Packet Loss. From our experimentation and detailed analysis there is significant overhead in Docker Swarm than in Kubernetes for CPU Utilization and Network Traffic. For Memory Utilization, this is opposite. Packet Loss resulted in 0 percent as network transfer is within the same network. When application is scaled, the percent of resource utilization increase in Kubernetes is higher when compared to Docker Swarm, but overall resource utilization of Kubernetes is much lower than that of Docker Swarm. By considering all the metrics and providing evidence that numbers obtained in this thesis are statistically significant and not by fluctuations(ANOVA and post-hoc analysis), we can better recommend Kubernetes over Docker Swarm for Web based Real Time Communication.

Kubernetes makes more efficient use of hardware to minimise resources needed to run containerised applications. Its automatic scalability is much efficient than Docker Swarm. It also scales resources as and when required with ease. Due to its broad usage, it has complex configuration. It requires separate set of administrative tools and constant upgrades coming from the Kubernetes community. Not all applications need the complex deployment, scheduling, and scaling services (or the overhead) that Kubernetes offers. But to meet the increasing demand for seamless Real time communications, and to suffice user requirements, the overhead offered by it is acceptable.

7.2 Future Work

In this thesis, we implemented an environment to orchestrate the WebRTC based Kurento server and Video conferencing application in both Kubernetes and Docker Swarm on master nodes. For experimental purposes, STUN/TURN servers were not required. It would be interesting to study the impact of performance of Kurento server across the other networks along with enabling the auto scale features and clustering(addition of worker nodes) of Kubernetes and Docker Swarm. This thesis only focused on performance in terms of CPU, Memory, Network Traffic and Packet Loss. So, evaluating the other metrics like Latency in all use cases is appealing. The same work can be further extended to study the impact in Cloud based environments(Single and Multi-Cloud Architectures) experimenting with N routing Mesh calls.

Also, thesis is confined with scaling the One-to-one application upto 14 users and One-to-Many application with 8 users due to system configuration. It could be really intriguing to study the Kurento server performance for hundreds and thousands of users. Similar research is presented in [60] but involving Kubernetes and Docker swarm or any other orchestration tool can affect those results.

8 REFERENCES

- [1] B. Garcia, L. Lopez-Fernandez, F. Gortazar, and M. Gallego, “Analysis of Video Quality and End-to-End Latency in WebRTC,” in 2016 IEEE Globecom Workshops (GC Wkshps), Dec. 2016, pp. 1–6. doi: 10.1109/GLOCOMW.2016.7848838.
- [2] E. Casalicchio and V. Perciballi, “Measuring Docker Performance: What a Mess!!!,” in Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, New York, NY, USA, 2017, pp. 11–16. doi: 10.1145/3053600.3053605.
- [3] C. C. Spoiala, A. Calinciuc, C. O. Turcu, and C. Filote, “Performance comparison of a WebRTC server on Docker versus virtual machine,” in 2016 International Conference on Development and Application Systems (DAS), May 2016, pp. 295–298. doi: 10.1109/DAAS.2016.7492590.
- [4] P. P. Ray, “An Introduction to Dew Computing: Definition, Concept and Implications,” IEEE Access, vol. 6, pp. 723–737, 2018, doi: 10.1109/ACCESS.2017.2775042.
- [5] “What is Cloud Computing,” Amazon Web Services, Inc. <https://aws.amazon.com/what-is-cloud-computing/> (accessed May 01, 2021).
- [6] Y. Xing and Y. Zhan, “Virtualization and Cloud Computing,” in Future Wireless Networks and Information Systems, Berlin, Heidelberg, 2012, pp. 305–312.
- [7] M. García-Valls, T. Cucinotta, and C. Lu, “Challenges in real-time virtualization and predictable cloud computing,” J. Syst. Archit., vol. 60, no. 9, pp. 726–740, Oct. 2014, doi: 10.1016/j.sysarc.2014.07.004.
- [8] K. Kumar and M. Kurhekar, “Economically Efficient Virtualization over Cloud Using Docker Containers,” in 2016 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM), Oct. 2016, pp. 95–100. doi: 10.1109/CCEM.2016.025.
- [9] “What is container orchestration?” <https://www.redhat.com/en/topics/containers/what-is-container-orchestration> (accessed May 02, 2021).
- [10] “What is Container Orchestration? Definition & Related FAQs,” Avi Networks. <https://avinetworks.com/glossary/container-orchestration/> (accessed May 02, 2021).
- [11] “Container Orchestration,” VMware. <https://www.vmware.com/topics/glossary/content/container-orchestration> (accessed May 02, 2021).
- [12] Y. Pan, I. Chen, F. Brasileiro, G. Jayaputera, and R. Sinnott, “A Performance Comparison of Cloud-Based Container Orchestration Tools,” in 2019 IEEE International Conference on Big Knowledge (ICBK), Nov. 2019, pp. 191–198. doi: 10.1109/ICBK.2019.00033.
- [13] “Documentation - Kurento.” <http://www.kurento.org/documentation> (accessed Mar. 24, 2019).
- [14] “Docker overview,” Docker Documentation, May 03, 2021. <https://docs.docker.com/get-started/overview/> (accessed May 03, 2021).
- [15] “What is a CaaS? Containers as a Service, Defined,” Rancher Labs, 45:23 - -0700 700. <https://rancher.com/caas-containers-service-defined/> (accessed May 03, 2021).
- [16] “Why PaaS vs. CaaS Is the Wrong Question to Ask,” Container Journal, Sep. 19, 2017. <https://containerjournal.com/features/paas-vs-caas-wrong-question-ask/> (accessed May 03, 2021).
- [17] “What is Real-Time Communications (RTC)?,” SearchUnifiedCommunications. <https://searchunifiedcommunications.techtarget.com/definition/real-time-communications> (accessed May 03, 2021).
- [18] “WebRTC,” WebRTC. <https://webrtc.org/> (accessed May 09, 2021).
- [19] “Kubernetes vs. Docker Swarm: What’s the Difference?,” The New Stack, Sep. 03, 2018. <https://thenewstack.io/Kubernetes-vs-docker-swarm-whats-the-difference/> (accessed May 04, 2021).
- [20] “Concepts,” Kubernetes. <https://Kubernetes.io/docs/concepts/> (accessed May 05, 2021).
- [21] G. Turin, A. Borgarelli, S. Donetti, E. B. Johnsen, S. L. Tapia Tarifa, and F. Damiani, “A Formal Model of the Kubernetes Container Framework,” in Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles, 2020, pp. 558–577. doi: 10.1007/978-3-030-61362-4_32.
- [22] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, “A Kubernetes controller for managing the availability of elastic microservice based stateful applications,” J. Syst. Softw., vol. 175, p. 110924, May 2021, doi: 10.1016/j.jss.2021.110924.

- [23]“Kubernetes Autoscaling in Production: Best Practices for Cluster Autoscaler, HPA and VPA.” <https://www.replex.io/blog/Kubernetes-in-production-best-practices-for-cluster-autoscaler-hpa-and-vpa> (accessed May 05, 2021).
- [24]F. Rossi, V. Cardellini, and F. L. Presti, “Hierarchical Scaling of Microservices in Kubernetes,” in 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), Aug. 2020, pp. 28–37. doi: 10.1109/ACSOS49614.2020.00023.
- [25]“Kubernetes Documentation.” <https://kubernetes.io/docs/home/> (accessed Mar. 24, 2019).
- [26]N. Naik, “Building a virtual system of systems using Docker Swarm in multiple clouds,” in 2016 IEEE International Symposium on Systems Engineering (ISSE), Oct. 2016, pp. 1–3. doi: 10.1109/SysEng.2016.7753148.
- [27]“Networking with overlay networks,” Docker Documentation, May 03, 2021. <https://docs.docker.com/network/network-tutorial-overlay/> (accessed May 05, 2021).
- [28]“Docker Documentation,” Docker Documentation, May 03, 2021. <https://docs.docker.com/> (accessed May 05, 2021).
- [29]“Autoscaling in Docker Swarm - Docker Hub,” Docker Forums, Jan. 12, 2018. <https://forums.docker.com/t/autoscaling-in-docker-swarm/44353> (accessed May 06, 2021).
- [30]“Auto-Scaling Services Using Instrumented Metrics - Docker Flow Monitor.” <https://monitor.dockerflow.com/auto-scaling/> (accessed May 06, 2021).
- [31]“Empowering App Development for Developers | Docker.” <https://www.docker.com/> (accessed May 09, 2021).
- [32]Prometheus, “Overview | Prometheus.” <https://prometheus.io/docs/introduction/overview/> (accessed May 07, 2021).
- [33]L. Lopez et al., “Kurento: The WebRTC Modular Media Server,” 2016, pp. 1187–1191. doi: 10.1145/2964284.2973798.
- [34]“Kurento.” <http://www.kurento.org/> (accessed May 09, 2021).
- [35]“Swarm mode overview,” Docker Documentation, May 03, 2021. <https://docs.docker.com/engine/swarm/> (accessed May 05, 2021).
- [36]“Deploy to Swarm,” Docker Documentation, May 06, 2021. <https://docs.docker.com/get-started/swarm-deploy/> (accessed May 09, 2021).
- [37]“GKE overview | Kubernetes Engine Documentation,” Google Cloud. <https://cloud.google.com/Kubernetes-engine/docs/concepts/Kubernetes-engine-overview> (accessed May 09, 2021).
- [38]“Amazon ECS | Container Orchestration Service | Amazon Web Services,” Amazon Web Services, Inc. <https://aws.amazon.com/ecs/> (accessed May 04, 2021).
- [39]mlearned, “Introduction to Azure Kubernetes Service - Azure Kubernetes Service.” <https://docs.microsoft.com/en-us/azure/aks/intro-Kubernetes> (accessed May 09, 2021).
- [40]“Browser APIs and Protocols: WebRTC - High Performance Browser Networking (O’Reilly),” High Performance Browser Networking. <https://hpbnc.co/webrtc/> (accessed May 06, 2021).
- [41]S. Loreto and S. P. Romano, “Real-Time Communications in the Web: Issues, Achievements, and Ongoing Standardization Efforts,” IEEE Internet Comput., vol. 16, no. 5, pp. 68–73, Sep. 2012, doi: 10.1109/MIC.2012.115.
- [42]“Web Real-Time Communications (WebRTC) transforms the communications landscape; becomes a World Wide Web Consortium (W3C) Recommendation and multiple Internet Engineering Task Force (IETF) standards.” <https://www.w3.org/2021/01/pressrelease-webrtc-rec.html.en> (accessed May 06, 2021).
- [43]“Signaling and video calling - Web APIs | MDN.” https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling (accessed May 06, 2021).
- [44]S. D. P. July 23rd, 2012 “Get Started with WebRTC - HTML5 Rocks,” HTML5 Rocks - A resource for open web HTML5 developers. <https://www.html5rocks.com/en/tutorials/webrtc/basics/> (accessed May 06, 2021).
- [45]“WebRTC - Signaling - Tutorialspoint.” https://www.tutorialspoint.com/webrtc/webrtc_signaling.htm (accessed May 09, 2021).

- [46] B. Garcia, L. Lopez-Fernandez, M. Gallego, and F. Gortazar, "Kurento: The Swiss Army Knife of WebRTC Media Servers," *IEEE Commun. Stand. Mag.*, vol. 1, no. 2, pp. 44–51, 2017, doi: 10.1109/MCOMSTD.2017.1700006.
- [47] L. L. Fernández, M. P. Díaz, R. B. Mejías, F. J. López, and J. A. Santos, "Kurento: a media server technology for convergent WWW/mobile real-time multimedia communications supporting WebRTC," in 2013 IEEE 14th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM), Jun. 2013, pp. 1–6. doi: 10.1109/WoWMoM.2013.6583507.
- [48] "Welcome to Kurento — Kurento 6.16.1-dev documentation." <https://doc-kurento.readthedocs.io/en/latest/> (accessed May 09, 2021).
- [49] E. Truyen, D. V. Landuyt, D. Preuveneers, B. Lagaisse, and W. Joosen, "A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks," *Appl. Sci.*, vol. 9, no. 5, p. 931, Mar. 2019, doi: 10.3390/app9050931.
- [50] A. M. Joy, "Performance comparison between Linux containers and virtual machines," in 2015 International Conference on Advances in Computer Engineering and Applications, Mar. 2015, pp. 342–346. doi: 10.1109/ICACEA.2015.7164727.
- [51] B. Jansen, T. Goodwin, V. Gupta, F. Kuipers, and G. Zussman, "Performance Evaluation of WebRTC-based Video Conferencing," *SIGMETRICS Perform Eval Rev*, vol. 45, no. 3, pp. 56–68, Mar. 2018, doi: 10.1145/3199524.3199534.
- [52] "Kubernetes vs Docker Swarm," *JRebel by Perforce*. <https://www.jrebel.com/blog/Kubernetes-vs-docker-swarm> (accessed Sep. 11, 2021).
- [53] V. Friðriksson, Container overhead in microservice systems. 2018. Accessed: Sep. 11, 2021. [Online]. Available: <http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-235693>
- [54] K. Wuestkamp, "K8s Monitor Pod CPU and memory usage with Prometheus," *Medium*, Sep. 23, 2020. <https://itnext.io/k8s-monitor-pod-cpu-and-memory-usage-with-prometheus-28eec6d84729> (accessed Sep. 11, 2021).
- [55] V. Sanagari, Live updates in High-availability (HA) clouds. 2018. Accessed: Sep. 11, 2021. [Online]. Available: <http://urn.kb.se/resolve?urn=urn:nbn:se:bth-16933>
- [56] "Troubleshooting Issues — Kurento 6.16.1-dev documentation." <https://doc-kurento.readthedocs.io/en/latest/user/troubleshooting.html> (accessed Sep. 11, 2021).
- [57] N. Marathe, A. Gandhi and J. M. Shah, "Docker Swarm and Kubernetes in Cloud Computing Environment," 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI), 2019, pp. 179-184, doi: 10.1109/ICOEI.2019.8862654.
- [58] M. A. Jawarneh et al., "Container Orchestration Engines: A Thorough Functional and Performance Comparison," *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, 2019, pp. 1-6, doi: 10.1109/ICC.2019.8762053.
- [59] "One-way analysis of variance," *Wikipedia*. Jun. 09, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=One-way_analysis_of_variance&oldid=1027745180
- [60] "How Many Sessions Can a Kurento Server Hold?," *testRTC*, Sep. 13, 2017. <https://testrtc.com/sessions-kurento-server/> (accessed Oct. 10, 2021).

9 APPENDIX

9.1 Monitoring with Prometheus

Prometheus offered monitoring of orchestration tools at various levels. We can monitor at node level, cluster level, container level to name a few. Below is the YAML file used to monitor Kubernetes.

```
global:
  scrape_interval: 15s # By default, scrape targets every 15seconds.
  external_labels:
    monitor: 'codelab-monitor'
# Scraping Prometheus itself
scrape_configs:
- job_name: 'prometheus'
  scrape_interval: 5s
  static_configs:
  - targets: ['localhost:9090']
- job_name: 'kubernetes-service-endpoints'
  kubernetes_sd_configs:
  - role: endpoints
  relabel_configs:
  - action: labelmap
    regex: __meta_kubernetes_service_label_(.+)
```

```
- source_labels: [__meta_kubernetes_namespace]
  action: replace
  target_label: kubernetes_namespace
- source_labels: [__meta_kubernetes_service_name]
  action: replace
  target_label: kubernetes_name
- job_name: 'kube-state-metrics'
  static_configs:
  - targets: ['kube-state-metrics.kube-system.svc.cluster.local:8080']
```

Figure 9.1 Prometheus YAML file of Kubernetes

Even though my scope is limited to Kurento media server container monitoring, I was able to monitor application container in the graphical interface exposed through services. Application container demonstrated high traffic than Kurento media server. Prometheus has the flexibility to choose between all the containers to monitor various fields.

9.2 ANOVA Post-hoc (Standard error differences) Test Results

One-to-One Application with 2 users

CPU Utilization:

Groups	Average
No Orchestration	7.946222
Kubernetes	0.082474915
Docker swarm	7.815679594

A	B	ABS(AVG(A)-AVG(B))
No Orchestration	Kubernetes	7.863747085
No Orchestration	Docker swarm	0.130542406
Kubernetes	Docker swarm	7.733204679

MS(Within Groups)	0.161841609
Standard Error	0.103872232

Memory Utilization:

Groups	Average
No Orchestration	0.4548889
Kubernetes	31.43479761
Docker swarm	12.35324726

A	B	ABS(AVG(A)-AVG(B))
No Orchestration	Kubernetes	30.97990871
No Orchestration	Docker swarm	11.89835836
Kubernetes	Docker swarm	19.08155036

MS(Within Groups)	205.7546883
Standard Error	3.703644046

Network Traffic:

Groups	Average
No Orchestration	293.7
Kubernetes	0.008701794
Docker swarm	295.0498207

A	B	ABS(AVG(A)-AVG(B))
No Orchestration	Kubernetes	293.6912982
No Orchestration	Docker swarm	1.349820657
Kubernetes	Docker swarm	295.0411189

MS(Within Groups)	1893.526672
Standard Error	11.23543998

One-to-One Application with 14 users

CPU Utilization:

Groups	Average
No Orchestration	13.515939
Kubernetes	0.214091979
Docker swarm	14.4671355

A	B	ABS(AVG(A)-AVG(B))
No Orchestration	Kubernetes	13.30184702
No Orchestration	Docker swarm	0.951196499
Kubernetes	Docker swarm	14.25304352

MS(Within Groups)	12.13199346
Standard Error	0.899332844

Memory Utilization:

Groups	Average
No Orchestration	0.5723889
Kubernetes	12.6988312
Docker swarm	3.086505411

A	B	ABS(AVG(A)-AVG(B))
No Orchestration	Kubernetes	12.1264423
No Orchestration	Docker swarm	2.514116511
Kubernetes	Docker swarm	9.612325786

MS(Within Groups)	177.728926
Standard Error	3.442178825

Network Traffic:

Groups	Average
No Orchestration	712.4603
Kubernetes	0.517640768
Docker swarm	639.9850089

A	B	ABS(AVG(A)-AVG(B))
No Orchestration	Kubernetes	711.9426592
No Orchestration	Docker swarm	72.47529107
Kubernetes	Docker swarm	639.4673682

MS(Within Groups)	26255.72829
Standard Error	41.83756549

One-to-Many Application with 2 users

CPU Utilization:

Groups	Average
No Orchestration	4.253111333
Kubernetes	0.102357106
Docker swarm	5.681288339

A	B	ABS(AVG(A)-AVG(B))
No Orchestration	Kubernetes	4.150754227
No Orchestration	Docker swarm	1.428177006
Kubernetes	Docker swarm	5.578931233

MS(Within Groups)	0.253623116
Standard Error	0.130031564

Memory Utilization:

Groups	Average
No Orchestration	0.401777767
Kubernetes	4.714091731
Docker swarm	1.595003569

A	B	ABS(AVG(A)-AVG(B))
No Orchestration	Kubernetes	4.312313964
No Orchestration	Docker swarm	1.193225802
Kubernetes	Docker swarm	3.119088162

MS(Within Groups)	2.629624487
Standard Error	0.418698339

Network Traffic:

Groups	Average
No Orchestration	148.6333333
Kubernetes	0.047146025
Docker swarm	153.6182747

A	B	ABS(AVG(A)-AVG(B))
No Orchestration	Kubernetes	148.5861873
No Orchestration	Docker swarm	4.984941352
Kubernetes	Docker swarm	153.5711287

MS(Within Groups)	218.9545136
Standard Error	3.820597803

One-to-Many Application with 8 users

CPU Utilization:

Groups	Average
No Orchestration	6.359888333
Kubernetes	0.187665375
Docker swarm	7.026439241

A	B	ABS(AVG(A)-AVG(B))
No Orchestration	Kubernetes	6.172222958
No Orchestration	Docker swarm	0.666550907
Kubernetes	Docker swarm	6.838773866

MS(Within Groups)	1.428108573
Standard Error	0.308556702

Memory Utilization:

Groups	Average
No Orchestration	0.4505
Kubernetes	1.789574463
Docker swarm	3.492656176

A	B	ABS(AVG(A)-AVG(B))
No Orchestration	Kubernetes	1.339074463
No Orchestration	Docker swarm	3.042156176
Kubernetes	Docker swarm	1.703081713

MS(Within Groups)	6.014040007
Standard Error	0.633195073

Network Traffic:

Groups	Average
No Orchestration	343.5
Kubernetes	0.19392788
Docker swarm	340.3452479

A	B	ABS(AVG(A)-AVG(B))
No Orchestration	Kubernetes	343.3060721
No Orchestration	Docker swarm	3.154752124
Kubernetes	Docker swarm	340.15132

MS(Within Groups)	15041.84549
Standard Error	31.6668549

