UNIVERSITY OF THESSALY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

MSC IN SCIENCE AND TECHNOLOGY OF ELECTRICAL AND COMPUTER ENGINEERING

Master Thesis

Implementation and Optimization of Real-Time Networking Frameworks for Containerized Remote Access Applications

Dallas Dimitrios

Supervising Professor:
Korakis Athanasios

Volos, Greece, 2023

UNIVERSITY OF THESSALY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

MSC IN SCIENCE AND TECHNOLOGY OF ELECTRICAL AND COMPUTER ENGINEERING

Master Thesis

Implementation and Optimization of Real-Time Networking Frameworks for Containerized Remote Access Applications

Dallas Dimitrios

Supervising Professor:
Korakis Athanasios

Volos, Greece, 2023

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Π.Μ.Σ. ΕΠΙΣΤΗΜΗ & ΤΕΧΝΟΛΟΓΙΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Μεταπτυχιακή Εργασία

Υλοποίηση και Βελτιστοποίηση Δικτυακών Πλαισίων Πραγματικού Χρόνου για Εφαρμογές Απομακρυσμένης Πρόσβασης σε Απομονωμένα Περιβάλλοντα

Δάλλας Δημήτριος

Επιβλέπων Καθηγητής:
Κοράκης Αθανάσιος

Βόλος, Ελλάδα, 2023

Approved by the Examination Committee:

Supervisor:	**Korakis Athanasios**
	Professor, Department of Electrical and Computer Engineering
	University of Thessaly
Member:	**Bargiotas Dimitrios**
	Professor, Department of Electrical and Computer Engineering
	University of Thessaly
Member:	**Antonios Argyriou**
	Associate professor, Department of Electrical and Computer Engineering
	University of Thessaly

Date of approval: 28/06/2023

ACKNOWLEDGEMENTS

First and foremost I would like to thank my family. I am truly grateful to my parents for being role figures in my childhood and for supporting me in my early adulthood. I am sure I can rely on them in times of need. My sister was a great part of my life from when I can remember myself and I am truly thankful for her standing by me all this time. I also would like to show my appreciation and love to my dear life partner for comforting me on bad days and making me smile on all of them.

In regards to my academic career and the completion of my academic studies, I want to express my deepest gratitude to professor Korakis Athanasios. His contribution and guidance to this project but more importantly to my development has been crucial since my undergraduate studies, and for that he has frankly my respect and appreciation.

Finally, I would like to thank all my fellow friends, colleagues and coworkers at the NITLab research facility for providing a friendly and more than pleasant working environment during the past years. Special thanks to Konstantinos Chounos, postdoctoral researcher at NITLab, for his most valuable contribution and assistance on the current master thesis.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this MSc thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of 8 intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

The Declarant

Dimitrios Dallas

05/07/2023

ABSTRACT

The current thesis proposes an implementation and network-aware optimization for real-time frameworks providing remote access to containerized applications. Initially it presents a transition of a given remote educational toolkit framework from a monolithic architecture to a Kubernetes cluster architecture consisting of multiple basic microservices. Its core functionality is serving a great variety of GUI based experiments on demand of the user while utilizing Kubernetes Jobs dynamic creation and termination. Implementing a cluster on a testbed infrastructure required some adjustments to the Kubernetes components compared to a regular cloud based infrastructure, like the use of Ingress controllers and Reverse Proxy Servers. Towards achieving optimal Quality of Service for the user on the GUI application streaming, a network-aware extension for the native Kubernetes scheduling mechanism was implemented. Network conditions of each worker node are altered to emulate real network heterogeneity of cloud environments. The scoring algorithm takes into account the Round Trip Time of the link between every worker and a given user device. Evaluating the proposed framework, video streaming metrics are monitored on the microservices, examining scenarios for both native and extended scheduling. The results promise a 25.4% increase to the average Frames per Second and a 37.3% increase to average Encoding Quality for an network heterogeneous cluster of nodes.

ΠΕΡΙΛΗΨΗ

Η παρούσα μεταπτυχιακή διατριβή προτείνει την υλοποίηση ενός δικτυακού πλαισίου και την βελτιστοποίηση του βάσει δικτυακής επίγνωσης για την απομακρυσμένη προσπέλαση απομονωμένων εφαρμογών πραγματικού χρόνου. Αρχικά παρουσιάζει την διαδικασία μετατροπής ενός υπάρχοντος απομακρυσμένου διαδικαστικού πλαισίου εργαλείων, από μονολιθική αρχιτεκτονική συστήματος σε ένα σύστημα συμπλέγματος κατανεμημένης αρχιτεκτονικής πολλαπλών μικροϋπηρεσιών. Η κύρια λειτουργία του είναι η εξυπηρέτηση μιας μεγάλης ποικιλίας εικονικών πειραμάτων γραφικού περιβάλλοντος κατόπιν αιτήματος του χρήστη. Οι εφαρμογές αυτές δημιουργούνται και τερματίζονται δυναμικά μέσω την χρήσης Jobs του Kubernetes. Η υλοποίηση του συμπλέγματος σε μια ερευνητική υποδομή απαιτούσε αρκετές προσαρμογές στα δομικά του στοιχεία συγκριτικά με μια τυπική υποδομή νέφους, όπως η χρήση Ελεγκτών Εισόδου και Εξυπηρετητές Αντίστροφης Μεσολάβησης. Με στόχο την επίτευξη της βέλτιστης Ποιότητας Εξυπηρέτησης για τον χρήστη στην εφαρμογή γραφικού περιβάλλοντος, υλοποιήθηκε μία επέκταση δικτυακής επίγνωσης για τον εγγενή μηχανισμό δρομολόγησης του Kubernetes. Προκειμένου να μιμούνται την ανομοιογένεια των πραγματικών δικτύων στα περιβάλλοντα νέφους, οι συνθήκες δικτύου σε κάθε κόμβο εργάτη αλλοιώνονται εξ επίτηδες. Ο αλγόριθμος βαθμολόγησης της εκτεταμένης δρομολόγησης λαμβάνει υπ’ όψιν του τον χρόνο μετ’ επιστροφής του συνδέσμου που μοιράζονται ένας εργάτης και ένας χρήστης. Για την τελική αξιολόγηση του προτεινόμενου πλαισίου, εξετάζονται σενάρια τόσο για τον εγγενή μηχανισμό δρομολόγησης όσο και για τον επεκταμένο, στα οποία παρακολουθούνται κάποιες μετρήσεις σχετικές με ροή βίντεο στις μικροϋπηρεσίες. Τα πειραματικά αποτελέσματα υπόσχονται μια αύξηση 25.4% για τον μέσο όρο των καρέ ανά δευτερόλεπτο και μία αύξηση 37.3% για την μέση ποιότητα κωδικοποίησης για ένα σύμπλεγμα κόμβων με δικτυακή ανομοιογένεια.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS vi
DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS vii
ABSTRACT viii
ΠΕΡΙΛΗΨΗ ix
TABLE OF CONTENTS x
LIST OF FIGURES xi
LIST OF TABLES xii
LIST OF ABBREVIATIONS xiii
CHAPTER 1: INTRODUCTION 1
1.1: Background 1
1.2: Problem Statement 1
1.3: Objectives and Scope 2
1.4: Related Work 3
1.5: Structure of the Thesis 3
CHAPTER 2: METHODOLOGY 5
2.1: Overview 5
2.2: Remote Educational Toolkit Framework 5
2.2.1: Graphical User Interface (GUI) Applications 5
2.2.2: X Persistent Remote Applications (Xpra) 6
2.2.3: Docker Environment 9
2.2.4: Web Servers 11
2.3: Kubernetes 12
2.3.1: Pods 12
2.3.2: Workload Resources 13
2.3.3: Networking 15
2.3.4: Kubernetes System Architecture 16
2.4: Networking tools 17
2.4.1: Reverse Proxy Server 17
2.4.2: Traffic Control 18
2.4.3: Ping Round Trip Time 18
CHAPTER 3: PROPOSED FRAMEWORK 20
3.1: Overview 20
3.2: Infrastructure and Development tools 20
3.3: Initial approach 21
3.3.1: Containerizing Existing System 21
3.3.2: Cluster-local implementation 23

3.4: LAN architecture 28
3.4.1: Ingress Controller 29
3.4.2: Reverse-Proxy Service 29
3.5: Latency-Aware Scheduler Extension 31
3.5.1: Native Scheduling Workflow 32
3.5.2: Latency Measurement 33
3.5.3: IP Acquiring 34
3.5.4: Scheduler Extender Server 34
CHAPTER 4: EVALUATION 38
4.1: Overview 38
4.2: Evaluation Setup 38
4.2.1: Metrics and Tools 38
4.2.2: Setup Architecture 39
4.2.3: Scenarios Description 40
4.3: Experiment Results 42
4.3.1: Native Scheduling 42
4.3.2: Low Scoring Extended Scheduling 44
4.3.3: Medium Scoring Extended Scheduling 45
4.3.4: High Scoring Extended Scheduling 47
4.3.5: Evaluation Results 48
CHAPTER 5: CONCLUSION 49
5.1: Outputs and Contribution 49
5.2: Future Work 49
References 50

LIST OF FIGURES

Figure 2.1: Xpra HTML5 Client with GNU Radio Companion virtual experiment............. 9
Figure 2.2: Kubernetes Deployment and Jobs Workload resources...............................14
Figure 2.3: Kubernetes Deployment and DaemonSet Workload resources................... 14
Figure 2.4: Service agnostic communication with Client outside of Cluster................... 16
Figure 2.5: Kubernetes Cluster components of Master and Worker nodes....................17
Figure 2.6: Reverse Proxy Server traffic forwarding to different servers and clients......18
Figure 3.1: Existing system architecture and example... 22
Figure 3.2: Transitioning to containerized system architecture.......................................23
Figure 3.3: Distributed Architecture for client inside the Cluster.....................................27
Figure 3.4: Proposed Architecture for Users outside the Cluster................................. 31
Figure 3.5: Kubernetes Scheduling workflow.. 32
Figure 3.6: Scoring formula for an individual worker node.. 36
Figure 3.7: Latency-Aware Scheduling over LAN architecture...................................... 37
Figure 4.1: Evaluation phase setups architecture..40
Figure 4.2: Xpra instances distribution on Extender Scheduling scenarios....................42
Figure 4.3: FPS vs Time for Native Scheduling scenario.. 43
Figure 4.4: Quality vs Time for Native Scheduling scenario.. 43
Figure 4.5: FPS vs Time for Low Scale Extended Scheduling scenario.........................44
Figure 4.6: Quality vs Time for Low Scale Extended Scheduling scenario.....................44
Figure 4.7: FPS vs Time for Medium Scale Extended Scheduling scenario...................45
Figure 4.8: Quality vs Time for Medium Scale Extended Scheduling scenario.............. 46
Figure 4.9: FPS vs Time for High Scale Extended Scheduling scenario........................47
Figure 4.10: Quality vs Time for High Scale Extended Scheduling scenario..................47

LIST OF TABLES

Table 1.1: Overall System outputs for Native and Extended Scheduling........................50
Table 1.2: Extended Scheduling evaluation of the Proposed Framework...................... 50

LIST OF ABBREVIATIONS

GUI	Graphical User Interface
CPU	Central Processing Unit
GPU	Graphics Processing Unit
RET	Remote Educational Toolkit
LAN	Local Area Network
XPRA	X Persistent Remote Applications
RTT	Round Trip Time
STEM	Science Technology Engineering Math
K12	Kindergarten to 12th grade
GRC	GNU Radio Companion
OVTR	Open Visual Traceroute
SDR	Software Defined Radios
IP	Internet Protocol
3D	Three Dimensional
PNG	Portable Network Graphics
RGB	Red Green Blue
HTML	HyperText Markup Language
WSGI	Web Server Gateway Interface
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ID	Identification
CNI	Container Network Interface
SSL	Secure Sockets Layer

JSON	JavaScript Object Notation
REST	Representational State Transfer
NITOS	Network Implementation Testbed using Open Source platforms
PC	Personal Computer
WSL	Windows Subsystem for Linux
RF	Radio Frequency
ICMP	Internet Control Message Protocol
DHCP	Dynamic Host Configuration Protocol
RAM	Random Access Memory
LTS	Long Term Support
UUID	Universally Unique IDentifier
DNS	Domain Name System
RPS	Remote Proxy Server
URL	Uniform Resource Locator
FPS	Frames per Second
MiB	Mebibytes

CHAPTER 1: INTRODUCTION

1.1: Background

Cloud based infrastructures are the go-to solution for more and more applications and services nowadays. Software development is peaking day by day with the revolution which distributed computing has brought. Containerization of applications with frameworks like Docker, with platforms like Kubernetes for container orchestration, really paved the way for virtualization techniques on distributed environments. More commonly though they are utilized for microservices rather than complete GUI applications. Those are traditionally expected to run as desktop applications due to demanding resource utilization. Running them on cloud means CPU and GPU intensive tasks move from the client side to the services, but the visual components must still be available to the users. Utilities for remote desktop sharing and remote desktop access can and do provide an agile solution for serving applications otherwise non-distributable or not yet developed for cloud usage.

1.2: Problem Statement

One of the many advantages which cloud applications provide is the freedom to access them from everywhere. Transitioning a framework from a monolithic to a distributed architecture adds another level of fault tolerance to the system and also enables scaling it up. Combined resources are able to serve more clients according to their demands. Kubernetes, like many other orchestration tools, manage near optimally physical computing resources and energy efficiency when scheduling new microservices to the available distributed hosts.

Regarding network conditions awareness, the default scheduling of Kubrenetes has some inefficiencies, even though there are clearly defined and usable mechanisms to manage and optimize in-cluster network links and traffic routing. Dealing with network heavy tasks like video streaming, may cause users to experience degradations on the quality of the video and latency issues. In the case of remote controlling GUI-based applications, the problem is magnified considering the inseparable requirement of interactiveness and responsiveness for user actions and the full-duplex nature of client-server communication. The current study contributes a step towards addressing such a problem, proposing a framework for distributed remote control of GUI-based applications and enhancing it with a latency-aware scheduling extension for Kubernetes for improved streaming quality and framerate.

1.3: Objectives and Scope

Several challenges are presented with this task, starting with the transition of an existing monolithic architecture framework to a distributed scheme. Proper container configuration needs to be addressed while also redefining system-level interactions of the user. The Remote Educational Toolkit (RET) framework discussed, already handles the initialization and termination of individual and short-term GUI-based experiments on user demand. Developing a framework for the distributed system to seamlessly provide the same functionality is clearly an objective of this work.

In the phase of implementation of such a framework, it is important identifying and overcoming any challenge arising relevant to testbed infrastructure, cluster coordination, inter-container communication and application configurations. A testbed environment can benefit research and development of a system like this as much as it can introduce drawbacks which need to be overcome.

After completing the system for in-cluster testing and Local Area Network (LAN) usage, designing a latency-aware extension for the native Kubernetes scheduling mechanism is the final objective of this thesis. An approach effective for a remote control utility like Xpra but not quite tailored to it, targets to make optimal use of the network conditions between each client and any available worker node and finally validate the proposed framework. The scope of this work focuses widely on video streaming applications served from cloud computing but more precisely GUI-based applications in cooperation with remote control utilities.

1.4: Related Work

Summarize three articles

In “Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing Applications”, Santos J. et al, present the inefficiencies of an orchestration tool to manage resources in regard to network conditions of the devices and the network requirements of the resources. In their proposed framework, given a cluster of fifteen nodes and one master, the cluster is divided in several zones of nodes, and every node of each zone has a predefined RTT label. Their Network-Aware Scheduler (NAS) use those labels to schedule pods based on their application’s need and even use a bandwidth requirement label to help with the scoring. Their test results are validated with the implementation of a Random Scheduler as the base scenario and then their NAS framework evaluation promises some improvements on the network efficiency.

In “Network-Aware Container Placement in Cloud-Edge Kubernetes Clusters”, Marchese A. and Tomarchio O. address the limitations of the default Kubernetes scheduler in the context of Cloud-Edge environments. The authors propose an improvement of the default Kubernetes scheduler to make its placement decisions aware of the run-time cluster network conditions and the communication interactions between microservices. They also propose a custom de-scheduler that periodically monitors run-time network state and traffic exchanged between microservices and evicts running Pods if they can be rescheduled onto more suitable nodes.

These papers provide valuable insights into the capabilities and potential of Kubernetes as a container orchestration system, its role in future platform development, and the need for network-aware scheduling in Cloud-Edge environments.

1.5: Structure of the Thesis

This master thesis consists of 5 chapters. After the current Chapter 1, a complete technical background and methodology follows in Chapter 2, discussing the selected tools and techniques to approach the problem. Chapter 3 analyzes in detail the specifics of the proposed framework, the challenges revealed during implementation and the solutions towards the objectives. Evaluation of the proposed framework is presented thoroughly in Chapter 4, discussing the implications of derived results, while also adding some extra info on the implementation of the metrics monitoring system. Finally, the thesis is completed with Chapter 5, which concludes the scope of the design and evaluation of the system and discusses future directions for further research.

CHAPTER 2: METHODOLOGY

2.1: Overview

In this chapter, a great variety of many different tools and technologies are outlined; all necessary for the research procedure of this thesis to be carried out. The objective was to examine existing core methods and mechanics currently in use on the RET in order to establish a solid foundation for the distributed cluster implementation. Past the developing tools of the proposed framework, the methodology chapter partially focuses on complementary tools and technologies dedicated to evaluate its performance characteristics.

2.2: Remote Educational Toolkit Framework

The Remote Education Toolkit Framework is an experimental education toolkit platform which allows K12 students to learn basic as well as advanced STEM concepts through running experiments on a remote Virtual Machine. The toolkit provides a curriculum that consists of Mathematics (Algebra and Geometry), Science (Physics and Life Science), and Computer Science (Signals and Networks), each offering a variety of lesson plans, in accordance to the different grades of the students that could access it. The three disciplines blend into an educational package supporting interactive virtual lab experiments through the use of a conventional computer and a Web Browser.

2.2.1: Graphical User Interface (GUI) Applications

The virtual lab experiments offer the users of the RET the opportunity to practically apply and test the knowledge of their corresponding lesson plan through audiovisual demonstration. GUI applications are a fundamental medium providing both audiovisual stimuli and interactiveness, making them excellent tools for virtualized labs. The chosen applications to be presented in the labs are highly relevant with the primary concepts of STEM education and also served as case studies for this research. Those are GNU Radio Companion (GRC) and Open Visual Traceroute (Ovtr).

Being free and open-source software, both GRC and Ovtr offer high adaptability for research and development works like this. GRC provides blocks for signal processing through a toolkit developed with Python bindings for the cross-platform Qt GUI widgets toolkit. Originally used in cooperation with Software Defined Radio (SDR) devices, the ADALM Pluto SDR, all signals are virtually generated in order for the experiments to be completely simulated. Using that visual interface, many signal processing flow graphs were developed, corresponding to the variety of lesson plans which end users can see executed simply by the press of a button. The majority of those experiments allow configurations for user input and change the output of the flow graphs, but all of them have features for scrolling and zooming in or out in the application window.

OvTr makes use of IP packet traceroute information, such as the geographical path or the hops from one network location to another. It combines both a powerful network diagnostic tool and a visually appealing presentation of the traceroute, which within the context of the RET framework blends with Mathematics and Computer Science principles. It provides spherical navigation of a 3D representation of the globe and also keyboard and clipboard forwarding.

GUI applications hosted within containerized environments present peculiar challenges, because, typically, containers are tailored for server applications rather than interactive GUI applications.

2.2.2: X Persistent Remote Applications (Xpra)

Xpra Server

Xpra (or "X Persistent Remote Applications") is an open-source multi-platform remote display server and remote desktop software. Xpra operates by attaching to running X sessions (X11 forwardings) and can also "detach" and "re-attach" to these sessions later. This gives a kind of persistence akin to GNU Screen for terminal sessions, but for graphical applications. It effectively allows users to run remote X11 applications and manage them as though they were running locally.

Xpra's unique design allows it to handle higher latency connections and maintain the graphical user interface's responsiveness, which can be a common issue with traditional X11 forwarding. It does this by utilizing different compression and encoding techniques to optimize the data transfer.

- Compression:

  Xpra uses various compression techniques to reduce the amount of data needed to be sent over the network. This compression can be applied at the pixel level (such as PNG or WEBP compression) or applied to the entire screen's pixel data (such as video encodings like H264, VP9, etc.). The choice of compression is dynamic and depends on several factors, including network bandwidth and latency, areas of the screen that are changing (and how much they're changing), and so on.

- Encodings:

  Xpra dynamically adapts to the available network bandwidth and tries to optimize the user experience by using different encoding techniques for the graphical data. It can switch encoding schemes on the fly. The encoding can range from raw RGB data (for local networks) to highly compressed H.264 video encoding (for slow or high-latency networks). It also supports lossless encodings for text and other areas where clarity is crucial.

Xpra HTML5 Client

The Xpra HTML5 client allows users to interact with the Xpra server, using a web browser, leveraging the WebSocket protocol. This client implementation is particularly useful because it doesn't require any special software to be installed on the client machine except for a modern web browser. The actual software is a web server installed side by side with the Xpra server. The client then can be accessed from within the frontend application of the distributed system. In the context of the RET, the Xpra

HTML5 Client is embedded within an HTML iframe component on every virtual class. It is only visible though only after the user initializes an experiment.

When the Xpra HTML5 client connects to the Xpra server, it establishes a WebSocket connection and begins receiving drawing commands. These commands instruct the client on how to render the remote application within the browser window, or in the embedded window component. The client sends back user inputs (mouse, keyboard, etc.) to the Xpra server.

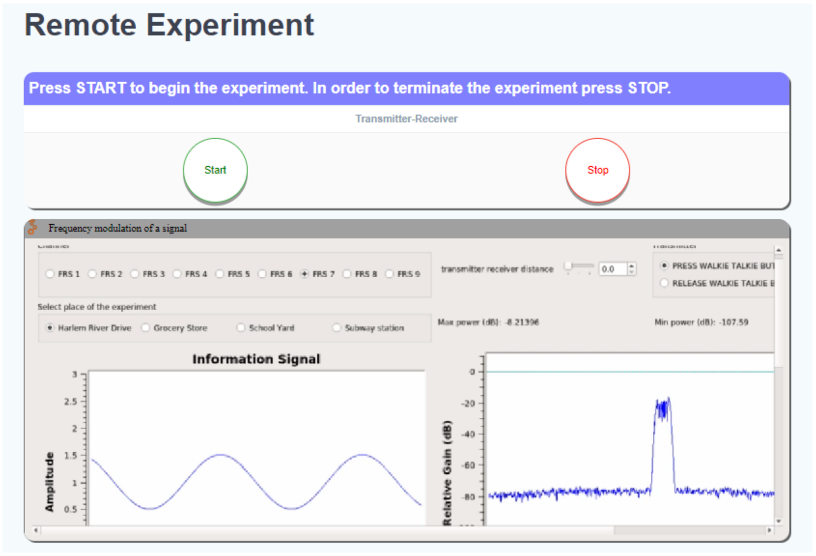
The Xpra HTML5 client can be customized with options such as scaling, encoding, quality, and speed, providing flexibility for the user's network conditions and requirements.

Metrics

Xpra has an integrated statistics module that provides various performance metrics of the Xpra server and the established connections. These metrics include details like damage events (how often and how much of the screen is updated), latency, compression and encoding speed, data sent/received, etc.

In the context of this research, a sidecar container was used to periodically fetch these metrics from the Xpra server using its control interface. These metrics are then analyzed to evaluate the impact of network conditions on the performance of the Xpra server and the user experience.

Integrating the Xpra utility in RET was a crucial feature because it enables the previously local desktop and CPU intensive desktop application to run on a remote machine. Users are able to experience the virtual labs just by receiving the audiovisual stream through the use of a conventional computer and a Web Browser, avoiding specific installations and powerful computing resources.



Press START to begin the experiment. In order to terminate the experiment press STOP.

Select place of the experiment: [✓ Harlem River Drive]

Channel: [✓ FRS 7]

transmitter receiver distance 0.0

[✓ PRESS WALKIE TALKIE BUT]

RELEASE WALKIE TALKIE B

Max power (dB): -8.21396

Min power (dB): -107.59

Figure 2.1: Xpra HTML5 Client with GNU Radio Companion virtual experiment

2.2.3: Docker Environment

Docker is an open-source platform designed to automate the deployment, scaling, and management of applications by using containerization. Containerization encapsulates an application along with its environment, making it highly portable and ensuring that it works uniformly across different computing environments.

Container Images

Docker container images are lightweight, standalone, and executable software packages including every necessary dependency needed to execute a piece of software, including the code, a runtime, libraries, environment variables, and config files. Images are immutable, meaning they do not change once created. This makes them ideal for version control and promoting changes from development environments to production.

Resource Management

Docker provides extensive resource management mechanisms that help control how much system resources a container can consume. Using Docker, you can specify limitations for CPU and memory usage, ensuring that a single container does not exhaust system resources. Docker also includes storage and network I/O controls, making it easier to manage resource consumption.

Networking

Docker creates its own network for inter-container communication. Each container gets its own network namespace, providing isolation from the host and other containers. Docker supports a variety of networking models, including bridge networks for isolation, host networking for performance, overlay networks for multi-host networking, and MACVLAN for routing network traffic to containers.

Port Forwarding

Docker is able to map any port within the container to a port on the Docker host. This port mapping allows the outside world to interact with the application inside the container as if it were running directly on the host itself.

Volume Sharing

Docker allows the sharing of data between the host and containers or between containers through the use of volumes. Docker volumes are managed by Docker and are designed to persist data independently of the container’s life cycle, thus offering a reliable way to store data generated by and used by Docker containers. Another use for them is to free some memory from each individual container’s file system by accessing some common and constant files directly from the host’s file system. This also serves the purpose of sharing periodically updated files like SSL certifications or systematically updated directories with new or debugged virtual labs.

Portability

Docker's containerization approach ensures that applications run the same no matter where they are or what machine they are running on. This uniformity simplifies the process of building, testing, and deploying applications across various environments, making Docker highly portable. Even though the selected hosting environments both for the RET and the current work are strictly linux based.

Integration with Xpra

Docker can be used to deploy applications along with Xpra. The appropriate application for each virtual lab and the Xpra utility can be packaged into individual Docker containers, for startup times and sizes proportional to the needs of the experiment. All files needed for the experiments and for Xpra connectivity are shared via volumes. Containers do not use their host networking but they occupy opportunistically a predetermined range of host ports with port forwarding. This setup allows the application to be run remotely via Xpra, with Docker handling the dependencies, isolation, and resource control.

Docker was already used to containerize the GUI-based applications and Xpra utilities in the core functionality of RET. In the context of this thesis it was also used to containerize the frontend and backend applications as well as any other software required for the challenges that appeared, facilitating their deployment on Kubernetes.

2.2.4: Web Servers

Flask is a lightweight web application micro-framework based on web server gateway interface (WSGI). Getting started is quick and easy by design of Flask, which is able to scale up to complex applications. Even though it offers suggestions, it require any change of dependencies or project layout.

Within the RET core functionality, Flask is used to create a publicly exposed API that handles HTTP/HTTPS requests and acts as an interface between the client and the Docker API. With each HTTP request from the client, Flask processes them, and triggers the creation or termination of individual Xpra containers. The response it generates are the ID of the container and the forwarded host port. With this information, the user can either access the Xpra HTML5 client within the container or trigger its termination.

Flask's simplicity and flexibility make it a good choice for such a task but also for extended uses within the frames of the proposed framework. It can be easily modified to access Kubernetes API rather than Docker API and is used to create easy and reliable remote functions like latency monitoring or a scheduler extension for Kubernetes.

Apache HTTP Server, colloquially known as Apache, is one of the most popular and widely used web servers in the world. Apache is recognized for its power, flexibility, and broad feature set, which includes a strong architecture for security and authorization, customizable log files, load balancing, and dynamic content negotiation. It was used to serve static and dynamic website content for the RET frontend, hosts the virtual labs files, the lesson plans and every educational material. It also serves as the web server of the proposed framework’s frontend application.

2.3: Kubernetes

Kubernetes is an open-source platform designed to automate deploying, scaling, and orchestrating application containers. During the implementation and evaluation of the proposed framework it was used as the orchestration platform for all microservices applications hosted on Docker containers. It is a powerful tool which allows advanced and complex system architectures, with many more components than a traditional monolithic system. Kubernetes has a great variety of components, all containerized software, either to ensure the functionality of the orchestration procedures or to enrich the features and capabilities of any orchestrated system.

2.3.1: Pods

A Pod is the simplest unit and the smallest entity in the scope of Kubernetes object model. Each Pod encapsulates one or multiple entangled application containers, storage resources, and are provided with a unique network IP, and configurations declaring a normal container execution. Containers within a Pod share an IP address and port space, and can communicate via localhost. They can also share storage volumes, allowing data exchange and communication between containers or even accessing the same application resources as if they run in the same host.

A Sidecar Container is a secondary, auxiliary container deployed inside a Pod to augment and enhance the functionality of the primary application container without changing the application itself. This pattern is crucial for tasks like logging, monitoring, or any functionality that we want to abstract away from the main application logic.

2.3.2: Workload Resources

Deployments

Kubernetes Deployments are used to declare a template for the desired application pods and ensure a specified number of pod replicas are running distributed across the worker nodes at any given time. On this template, everything is defined, from metadata and labels for the pods to the container image, exposed ports and volume mounts. A Deployment controller provides declarative updates for Pods according to the desired state. At any occurring change on the system, like a pod failure, the Deployment controller changes the actual state to the desired state at a controlled rate.

Secrets allow for the management of sensitive information, such as passwords, ssh keys, etc. Secrets are decoupled inside the Pod or Deployment definition and do not store its information within the application code.

Jobs

Jobs are another kind of controller like Deployments that creates one or more Pods and sees through successfully terminating a specified number of them. Jobs in Kubernetes are used to run finite tasks or batch processes. They ensure the completion of one or more tasks and are well-suited for computational tasks and setup tasks. Jobs are suited for running tasks to completion, like batch jobs or one-time configurations. And for that dynamic and temporary they fit perfectly for defining the Xpra Pods running the experiments on user demand.

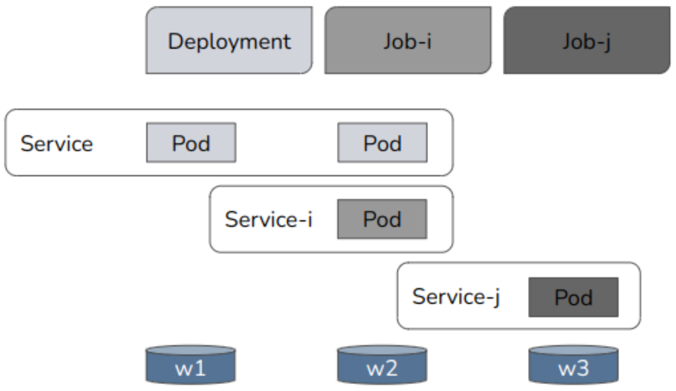


Figure 2.2: Kubernetes Deployment and Jobs Workload resources

Daemon Sets

DaemonSets are used to ensure that some or all nodes run a copy of a Pod, which can be useful for deploying system-wide services. With the addition of new nodes to the cluster, Pods are added to them. Upon their removal or failure, those Pods are garbage collected. This can be particularly useful for microservices required on the totality of the worker nodes, and needed to be deployed from their addition to the cluster.

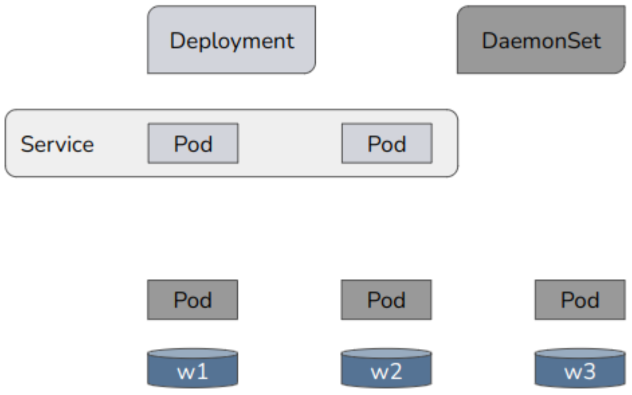


Figure 2.3: Kubernetes Deployment and DaemonSet Workload resources

2.3.3: Networking

Services

Services in Kubernetes are an abstract way to expose a specific kind of applications running on a set of Pods. The set of Pods targeted by a Service is usually determined by a selector. Services without selectors and those that target a specific Pod are also possible use-cases. Depending on the use case, Services can be exposed in different ways by specifying a type in the serviceSpec: ClusterIP, NodePort, LoadBalancer, and ExternalName.

Networking

Kubernetes provides several networking features that were vital to this project. Its flat networking model ensures every pod can communicate with every other pod and node in the cluster. It supports Network Plugins, and the Container Network Interface (CNI) is a specification that should be used by these plugins. Flannel is one such CNI compliant network plugin that provides a simple and easy way to configure a layer 3 network fabric designed for Kubernetes. Network policies can be defined to control the flow of traffic, while ingress and services expose applications to the outside world.

Ingress Controller

In the Kubernetes ecosystem, the Ingress Controller is an integral component that governs and manages external access to the services within a cluster. Essentially, it is responsible for directing external HTTP and HTTPS traffic to the appropriate internal services, based on a collection of rules defined in the Ingress resource. This controller is responsible for monitoring Ingress objects in the cluster and configuring a HTTP load balancer to route traffic accordingly. It also provides features like SSL termination, path-based routing, and host-based routing.

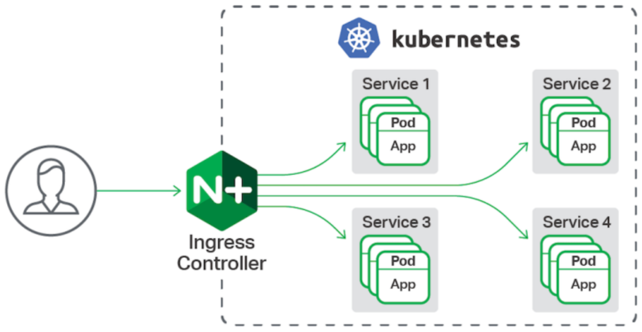


Figure 2.4: Service agnostic communication with Client outside of Cluster

(Source: [DevPress])

2.3.4: Kubernetes System Architecture

Kubernetes uses a client-server architecture, where its components can be divided into those that manage an individual node and those providing the cluster's control plane. The latter includes the API Server, Controller Manager, and Scheduler.

API Server

The API Server, or kube-apiserver, is the main management component of Kubernetes. It serves the Kubernetes API using JSON over HTTP, which provides both the internal and external interfaces to Kubernetes. It processes REST operations, validates them, and updates the corresponding objects in etcd, which is a consistent and highly-available key value store used as Kubernetes' backing store for all cluster data. The API Server is the front end for the control plane.

Controller Manager

The Controller Manager, or kube-controller-manager, runs the core control loops within Kubernetes. A control loop is a non-terminating loop that normalizes the state of a system. The controllers include the Node Controller, Replication Controller, Endpoints Controller and Service Account among others. These controllers observe the shared state of the cluster through the API Server and make changes in an attempt to regulate the current state towards the desired state.

Scheduler

The Kubernetes Scheduler, or kube-scheduler, keeps a scheduling queue for created Pods with no assigned node and selects a node to host and execute them. Factors taken into account for scheduling decisions include individual and collective resource requirements, hardware or policy constraints, affinity and anti-affinity specifications, data locality, and inter-workload interference. The Scheduler ensures that resources are utilized effectively while respecting defined constraints, leading to optimized workload performance.

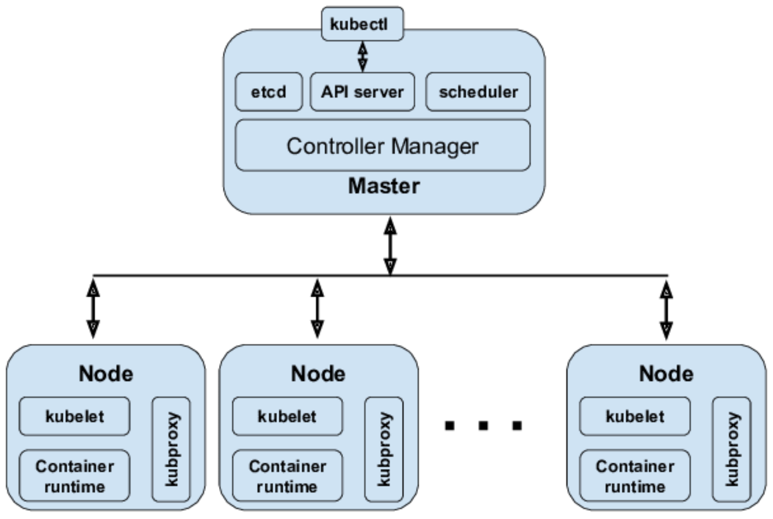


Figure 2.5: Kubernetes Cluster components of Master and Worker nodes

(Source: Andrew J. Younge)

2.4: Networking tools

2.4.1: Reverse Proxy Server

A reverse proxy server, a crucial architectural component in many network infrastructures, is a server that retrieves resources on behalf of a client from one or more servers. His counterpart, the forward proxy is an intermediary for its associated clients to contact any server, but a reverse proxy works as an intermediary for its associated servers to be contacted by any client. This effectively abstracts the network and provides security, load balancing, and caching functionalities. In this context, it can be considered a technique for ensuring that network traffic is efficiently and securely managed.

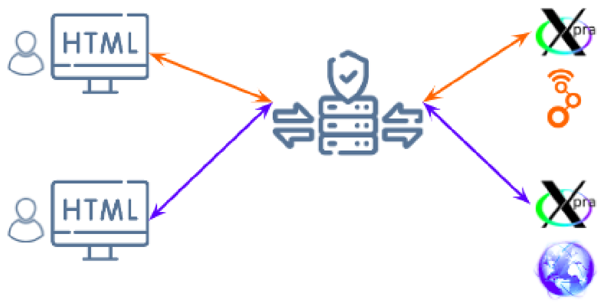


Figure 2.6: Reverse Proxy Server traffic forwarding to different servers and clients

A reverse proxy server acquires and collects resources on behalf of a client from one or more servers of his choosing. NGINX was used in this capacity, directing client requests to appropriate Xpra servers.

2.4.2: Traffic Control

Traffic control, in the context of network management, involves the process of managing and controlling network traffic to reduce congestion, latency, and packet loss. It includes techniques like bandwidth shaping, prioritization, and access control. In Kubernetes, it can be particularly useful for simulating different network conditions and testing how applications behave under those conditions. With the 'tc' (traffic control) utility in Linux, it can be used for creating conditions of network latency, control network bandwidth, delay, jitter, and packet loss in the network to measure the impact on Xpra's performance.

2.4.3: Ping Round Trip Time

Round trip time (RTT) is a critical metric in network performance, measuring the time it takes for a signal to travel from a source to a destination and back again. In essence, it is a gauge of the latency in a network. In applications that require real-time communication, such as video streaming or online gaming, low RTT is crucial for a good user experience. Tools like 'ping' or 'traceroute' can be used to measure RTT, providing insights about network performance and aiding in troubleshooting network issues.

A network diagnostic tool like ‘ping’ can be used to test if a host is reachable on an Internet Protocol (IP) network and to measure the RTT for packets sent from the originating host to a destination. It was used as a basic tool for network latency measurements.

CHAPTER 3: PROPOSED FRAMEWORK

3.1: Overview

In this chapter, the general procedure towards the implementation of the proposed framework is described. Small and steady steps are taken targeting the objectives of this thesis and challenges occur along the way. First, a brief description is presented for the research environment’s infrastructure and the development specific tools and commands which facilitated the current implementation. Following that are the transition steps towards the final architecture approach. And finally, on the stable and normally functioning Distributed RET, a scheduler extension mechanism is implemented and analyzed focusing on latency awareness and leads to evaluation of the system in the next chapter.

3.2: Infrastructure and Development tools

The entirety of the research, development and evaluation phase was exclusively carried out on the NITOS Testbed, deployed in the facilities of the University of Thessaly campus. NITOS supports experimentation-based research in the area of wired and wireless networks and is continuously available and remotely accessible. The nodes selected as workstations were part of the Indoor RF Isolated Testbed. The base workstation was a Windows 10 PC with WSL 2 for development and testing operations and the application interface with the proposed framework was a conventional web browser. An inbound firewall rule was added on windows defender to permit and respond to ICMP packet traffic.

Each node has natively enabled only one ethernet interface which interconnects them via their common isolated subnet. They are equipped, though, with another interface for ethernet connections via the University’s network. When enabled with the `dhclient` command, it receives an IP address from the gateway’s DHCP server and an IP route rule is added forwarding traffic towards the subnet of the base workstation. Because the evaluation of the proposed framework targets network heterogeneity, all nodes occupied were equipped with Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, two RAM devices of

8GiB DDR3 Synchronous 1600MHz and have operating system Ubuntu 18.04.2 LTS with 4.15.0-47-generic kernel. With those features and a configuration in their affinities, resource homogeneity is achieved so that the evaluation results are strictly focused on networking resources.

All of the tools previously mentioned in Chapter 2: Methodology played a crucial role in the implementation of the proposed framework. The command-line tools *kubectl, kubeadm* and *kubelet* are all in their GitVersion v1.20.0 and provide a stable environment for the cluster. Kubectl is the basic command that accesses the API Server of the Master node and makes requests regarding the resources of the cluster, either creating, terminating or inspecting them. Kubeadm is the tool which allows a master node to initialize a cluster with specific configurations and generates valid tokens and discovery information so that other nodes can join the cluster. Finally, kubelet is the main agent in the nodes responsible for managing the container runtime and making sure YAML and JSON definitions of objects are running properly. The container runtime is Docker version 19.03.6 which also allows the use of Dockerfile definitions. Dockerfiles were accountable for the creations of the multiple docker images used in this implementation and pushed to Docker Hub, the distributor from which each cluster node pulled the desired resources.

3.3: Initial approach

3.3.1: Containerizing Existing System

The first step towards creating an orchestrated distributed system was to disengage the core functionalities of the existing framework from their host. Their respective applications should convert from system processes to container applications in order to adapt in the cluster environment which requires software portability. So the existing system running on a Virtual Machine in a public server, was replicated with a fresh installation on a Testbed node. All running under the same device, Docker is the basic system process which needs to be executed directly under the node host.

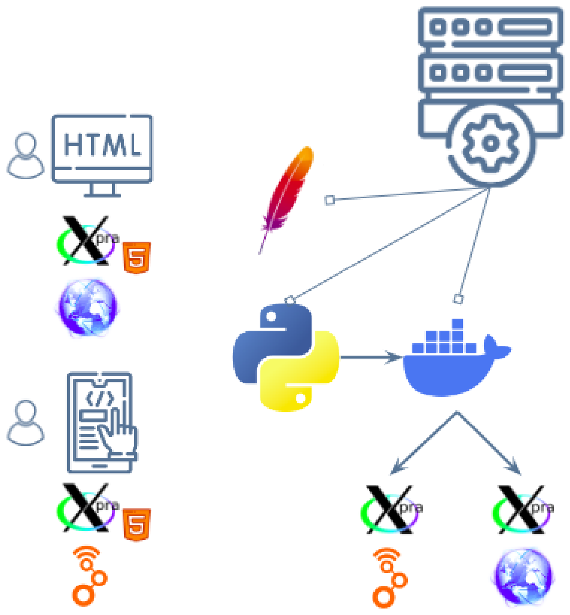


Figure 3.1: Existing system architecture and example

The backend server of the existing system is using the Python Docker API to access Docker functionalities and manage containers. That is the case because the Flask web server runs side by side with the Docker Engine, on the same host, and has direct access to Docker's daemon. This daemon listens to a Unix socket located on */var/run/docker.sock*. When containerizing the application, the Python module needs access to the host’s daemon so the chosen practice is to share the socket file with a mounted volume between host and container. This can be achieved on *docker run* command:

```
:~# docker run -d -p 8090:5000 -v
/var/run/docker.sock:/var/run/docker.sock backend_image
```

Other than the mounted volume with ‘-v’ argument, ‘-p’ enables port forwarding of the container internal port 5000 to host port 8090. Inside the backend server’s code it only needs the addition of the following snippet.

```
import docker
client = docker.from_env()
```

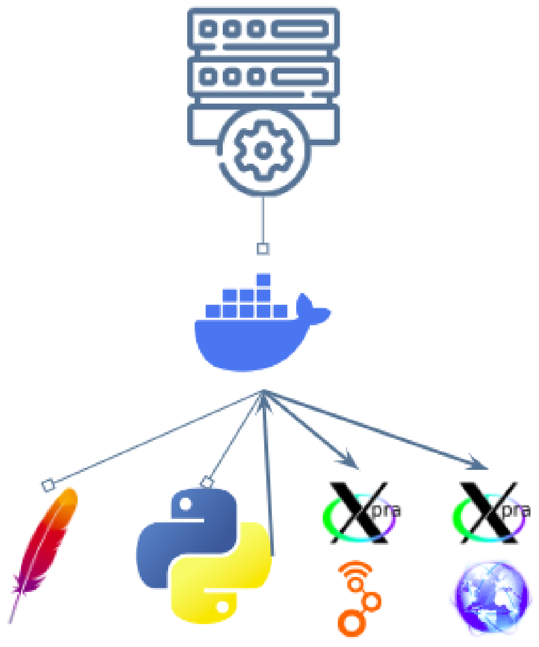


Figure 3.2: Transitioning to containerized system architecture

With this setup configuration, a client can have the same experience virtually even though the inner workings of the system processes are different. This transition is the very first step towards creating a distributed system. Running all core functionalities under a container runtime offers great portability and abstraction to a system. Now this new architecture of the host device is just an abstract architecture for every worker node in the proposed framework and now the system can move to a cluster approach.

3.3.2: Cluster-local implementation

Containerizing system processes lets the Docker Engine manage resources but even that practice can not fully exploit the available resources and deal with techniques like fault tolerance, scalability and availability. When dealing with a cluster system, resources are combined and so can the instances of the microservices, not strictly binded to the resources of a single host.

With concepts previously mentioned on Chapter 2: Methodology, the various applications of the existing system were integrated to workload resources of the Kubernetes system. Frontend and backend applications were assigned to Deployment definitions with appropriate application labels and a selected number of replicas on their respective YAML files. The Controller Manager of the Master node works to constantly provide the exact number of the desired replicas, after Pod failures or Node failures. Each template within the YAML files give a specific name for the container applications of the same Deployment, one or multiple port numbers which represents the internally exposed ports of the container application and finally declare the desired container image for the application. As stated previously, this image is pulled from the Docker Hub database with an optionally selected policy in the specification definition, like *IfNotPresent* or *Always*, in order for each individual node to pull the most updated image version.

Each Deployment, though, had its own distinctions to function properly. Because Docker Hub distributes publicly docker images, the frontend images were stored in a private repository within the stated platform, in order to secure the copyrighted material of the existing RET framework. For that reason, the frontend Deployment had the addition of *imagePullSecrets* field on the template specification, which makes use of the *Secret* resource of Kubernetes. That kind of resource extracts account authorization credentials, encrypted to base64 data and stored to a configuration file.

Settling to a cluster environment requires an alternative for the dynamic nature of the Xpra instances representing a virtual experiment. Kubernetes Jobs are exactly the workload resource controller needed for dynamically created and terminated pods. Usually, this applies for executions with a well defined goal or execution time, which can start, fulfill their purpose and terminate normally. Even though that is the case with many of the virtual experiments of the RET, some others run indefinitely until the user requests their termination. In case of an Xpra-pod failing, the Job controller creates a new one to replace it. Kubernetes Jobs have similar YAML definitions as Deployments with the addition of the *job-name* label.

For the backend template specification, the initial addition was the Docker daemon socket mounted volume, reflecting the initial approach of the containerized existing system. This had drawbacks of not fully utilizing cluster features because each deployed backend pod had access to the specific daemon socket of its own worker node host. So every backend pod which received HTTP requests for virtual experiments could only create Xpra container instances on his host environment and not Xpra Jobs, they were not actually inspected and controlled by the Kubernetes system. The best practice for the desired functionality was to replace the Docker module with the Kubernetes module in Python.

```python
from kubernetes import client, config, utils
import uuid, yaml, tempfile

# Configure in-cluster client
config.load_incluster_config()

backend_pod_uid = os.environ.get('BACKEND_POD_UID')

batch_v1 = client.BatchV1Api()
core_v1 = client.CoreV1Api()
k8s_client = client.api_client.ApiClient()
```

Loading the “in-cluster” configuration on the python runtime of the backend Pod, allows the creation of a kubernetes client on that runtime to communicate with the Kubernetes API server, and access it accordingly to an assigned ServiceAccount. This ServiceAccount resource, through the use of a RoleBinding resource, is combined with a Role resource. The Role resource is the one which clearly defines the api groups (batch, extensions, apps), resources (jobs, services) and verbs (create, delete) the desired ServiceAccount will have privileges for. The ServiceAccount field is filled inside the Backend Deployment definition.

Another feature mandatory to the distributed system’s backend server is the ability to initiate every possible virtual experiment available for a current user. On the existing RET system, the backend server constructed the appropriate execution command for the GUI applications and passed it to the Xpra container initialization as an entrypoint argument. All required files were resigning inside Xpra docker images. In order for the same feature to be replicated on Job creation and configuration rather than container initialization. This is why on the proposed framework, the backend pods contain YAML templates of the Xpra jobs definitions. On user demand, the backend extracts the requested information and replaces the appropriate fields inside the templates.

```python
def create_xpra_job(unique_id, version, start_command, client_ip):
  with open("xpra-job.yaml", 'r') as file:
    job_template = file.read()
  job_template = job_template.replace("{{unique_id}}", unique_id)
  job_template = job_template.replace("{{version}}", version)
  job_template = job_template.replace("{{start_command}}", start_command)
  # Create a temporary file
  with tempfile.NamedTemporaryFile(mode='w', suffix='.yaml') as temp_yaml_file:
    temp_yaml_file.write(job_template)
    temp_yaml_path = temp_yaml_file.name
    utils.create_from_yaml(k8s_client, temp_yaml_path)
```

In the above snippet of code, the backend server designates the *unique_id* of the Xpra-job, its *version* defining the docker image with the desired GUI application and finally *start_command* determines the particular files instructions for executing a specific experiment. A *unique_id* is just a generated Universal Unique Identifier (UUId) using its own backend-pod *metadata.uid* as prefix. This *metadata.uid* is passed as an environment variable on the Deployment definition and was developed as a fail-safe mechanism so that duplicate generated uuid will not occur, regardless of the UUId versioning. Once the customized temporary YAML file is ready, backend communicates with Kubernetes API Server and requests the desired resource to be created and scheduled.

After applying all those Deployments and Jobs to the Kubernetes cluster, the multiple or individual pods needed a resource to be discovered or reachable. Two constant ClusterIP services were applied for frontend and backend pods respectively. Services were also used for reaching each individual Xpra-pod created by a given *Xpra-job-uuid*. A backend-pod, using the same technique previously demonstrated for creating Xpra-jobs, requests from the API Server a corresponding *Xpra-service-uuid*, which has a selector field matching the job-name of *Xpra-job-uuid*. In that way, Xpra-jobs and Xpra-services have a one-on-one relationship. In a typical cloud based environment, the best practice would be to make use of a LoadBalancer Service selecting generally every Job with prefix *xpra-job*, and have it route traffic to the appropriate one.

```
def delete_xpra_job_and_service(unique_id):
  delete_options = client.V1DeleteOptions(propagation_policy="Background")
  batch_v1.delete_namespaced_job(name=f"xpra-job-{unique_id}", body=delete_options)
  core_v1.delete_namespaced_service(name=f"xpra-service-{unique_id}")
```

After a user has sufficiently executed his experiment, he sends a termination request to the backend server. The client sends the stored UUId value to the server and he requests the termination of the appropriate resources from the API Server.

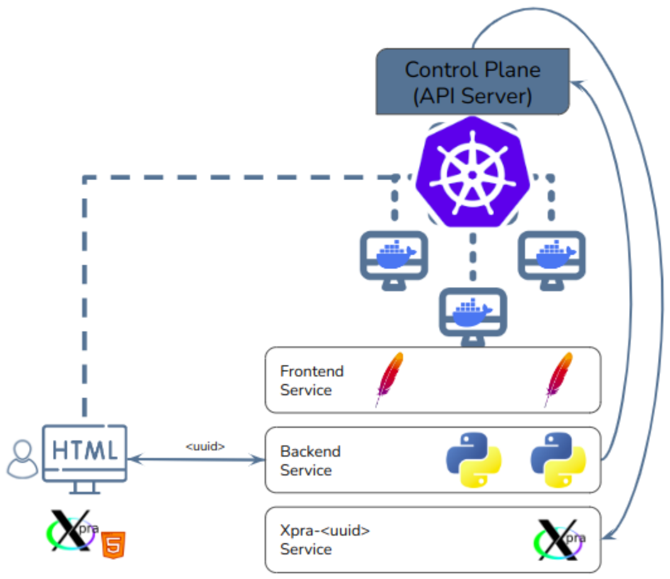


Figure 3.3: Distributed Architecture for client inside the Cluster

In this initial approach of clustering the existing system core functionalities, the testing client was originally a node device connected to the cluster but disengaged of any workload resources. The client could access every established service using its ClusterIP because he also is part of the cluster. But Xpra-services are dynamically created and the application level interaction of the client to the backend server would require a delay and data overhead. The backend server should request to inspect a freshly created xpra-service from the API Server to be informed of its assigned ClusterIP and then communicate it back to the client. Instead of that, on the user's initialization request, the backend server responds with the assigned UUId if the Xpra instance.

In this scenario it was best utilized the feature of DNS records of the Kubernetes system. Any cluster pod application can discover and communicate with another cluster pod or service with the user its DNS, being abstractly *name.namespace.object.cluster-domain.example*. Now with that approach, and all services applied in the default namespace, the frontend application could statically refer to its own address with *frontend.default.svc.cluster.local* and to the backend address with *backend.default.svc.cluster.local*. In a similar manner, with the use of a runtime variable in the frontend application, the client can dynamically change the requested address of an Xpra-service, accessing it through *xpra-<uuid>.default.svc.cluster.local.* This address is passed to the source attribute of the iFrame component of the HTML, and can be updated between user requests. The final issue to be addressed is the addition of the kube-dns nameserver in the */etc/resolv.conf* of the client node, because it is configured only internally in the pod system.

3.4: LAN architecture

With the implementation discussed above on Subsection 3.3.2, many of the core functionalities and features of the proposed framework were already deployed and serviceable. Optimally though, the desired setup for the proposed framework of this thesis and its evaluation requires a more cloud-like approach, meaning that the cluster system should be abstract and agnostic for each client communicating with it. The default assumption is that the client will not be part of the cluster and thus will only be able to reach it via a public IP or even a public DNS uri. Due to the current infrastructure of the NITOS Testbed, and not a paid plan in a cloud provider, this cloud-resembling setup is simulated within the Local Area Network (LAN) with the use of a known host and a Kubernetes Ingress Controller.

3.4.1: Ingress Controller

An Ingress Controller, as previously explained in Chapter 2: Methodology, is a Deployment of one or multiple replicas of reverse proxy servers using NGINX which is deployed over an according amount of worker nodes and is usually exposed with a LoadBalancer Service. Again, adapting to the Testbed infrastructure, the optimal service type for Ingress Controller should be NodePort. This type of service exposes a specific port across all worker nodes' primary interface and allows the controller pods deployed on a specific worker node to be accessed from outside the cluster. So a typical client reaching the NITOS worker nodes on LAN, can access an Ingress Controller pod from a given NodePort. For convenience of setup, only one worker node was selected to host only a single Ingress Controller pod.

This Ingress Controller receives requests from a client on LAN. The HTTP routing properties of it are defined through a set of rules called Ingress rules. Ingress rules do not expose specific endpoints, but rather declare which specific endpoint prefixes correspond to already existing cluster services. So the Ingress Controller parses every HTTP request it receives and forwards it according to the endpoint it tries to reach using the DNS records of the services. In the current architecture step, the Ingress rules know about frontend-service and backend-service. This comes with a major drawback which is the lack of being aware prematurely of the dynamically created xpra-services and their DNS records. In order for the client to be able to communicate with the xpra-services without being part of the cluster, the following Reverse Proxy Service technique is proposed.

3.4.2: Reverse-Proxy Service

After the initialization of an xpra-service with a given UUId, a DNS record is created on the name *xpra-service-<uuid>.default.svc.cluster.local* and every pod with the kube-dns resolver can access it on creation time. For the proposed framework component of the Ingress Controller to work, there needed to be a mechanism for dynamic traffic routing of HTTP requests targeting the xpra-instances. This mechanism works with any request reaching the Ingress Controller and does not contain neither a frontend prefix nor a backend prefix, this request will be forwarded to the DNS of Reverse Proxy Service (RPS), meaning *reverse-proxy.default.svc.cluster.local*, as defined by the Ingress rules. Therefore, for a supposed constant and unchanging xpra-service created and terminated on user demand, traffic is routed from the Ingress Controller to the RPS and from then to *xpra-service-<uuid>*. Traffic follows the same hops back to the client. The RPS is implemented with a standard NGINX image deployment with a volume mounted on */etc/nginx/conf.d* providing it a modified version of the *default.conf* file displayed below.

```
server {
  listen 80;
  resolver kube-dns.kube-system.svc.cluster.local valid=5s;
  location / {
    set $target "http://xpra-service-$cookie.default.svc.cluster.local:9876";
    proxy_pass $target;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  }
}
```

The above snippet shows that the RSP uses the *kube-dns* resolver in order to obtain the DNS of *xpra-service-<uuid>* and will keep that response for 5 seconds before requesting him again. The HTML5 client of an xpra-instance is requesting many resources which are common among all generic Xpra servers and there is no native way to distinguish them from their prefix on the Ingress rules or in RPS. In view of the fact that a single RPS could have workloads from multiple clients which have requested similar resources, the proposed framework benefits from the use of browser cookies. When a backend-pod responds to the UUId of an Xpra-instance to a client, this information is stored in a browser cookie for the sole purpose of appropriate traffic forwarding from an RPS to the desired Xpra-service. This cookie is simply used as a string variable within the DNS record of the targeted Xpra-service.

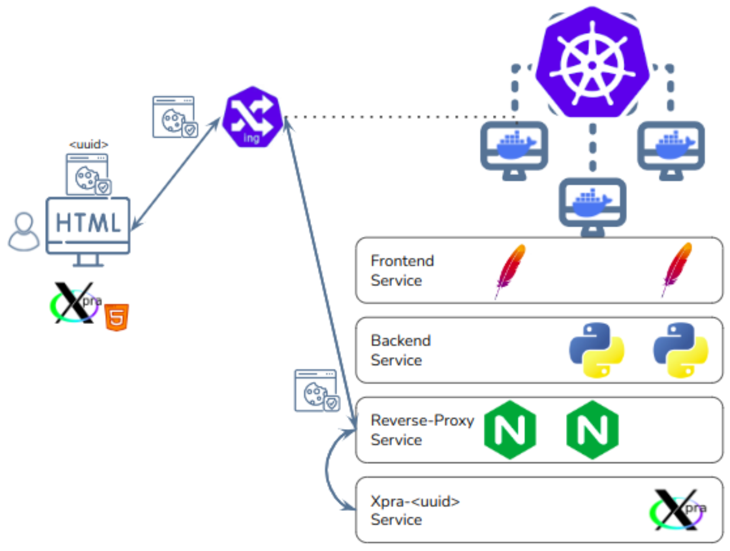


Figure 3.4: Proposed Architecture for Users outside the Cluster

In this architecture for LAN users, the proposed framework is fully functional and the transition is seamless from the existing RET framework. The client machine adds to its known host, under the hostname *ingress.controller*, the NodeIP of the worker hosting the Ingress Controller. In this way, within a conventional browser, every request resembling *http://ingress.controller:nodeport/endpoint* is reaching the Ingress Controller and is forwarded to frontend, backend and reverse-proxy services. For example *http://ingress.controller:nodeport/index.html* is forwarded to the frontend applications, *http://ingress.controller:nodeport/backend/create/experiment-x* is forwarder to the backend applications and every other request on the root endpoint is routed to the RPS applications.

3.5: Latency-Aware Scheduler Extension

With the above subsection, a stable system was implemented and deployed, functioning in a distributed manner and enables experimentation towards its optimization. Many directions were open for optimization on the current system but as stated in Chapter 1: Introduction, the objective of this thesis was to increase the general quality of a video streaming application, such as the Xpra remote desktop control application. In this context, an extension to the native Kubernetes scheduling process is proposed and developed with cluster components and resources. The objective of deploying a Network-Aware extension for scheduling is to bind pods in the scheduling queue on worker nodes which better serve the corresponding client. Searching and targeting for the worker-client link with minimum latency relative to the rest of the worker nodes is a mechanism which will result in general quality improvement on the Xpra applications.

3.5.1: Native Scheduling Workflow

The native scheduling workflow of Kubernetes consists of multiple stages and points, some of them extensible, some other not. The crucial and most easily comprehensible stages of said workflow is a 2-step operation, the *filtering* phase and the *scoring* phase. These two steps are basically responsible for assigning a requested pod to a promising worker node, in regard to the native scheduling rules. Initially a pod in the scheduling queue is selected and in the filtering phase, all worker nodes are examined through various conditions like NodeAffinities and those not suitable are filtered out. The node objects which pass the filtering test gradually end up in the scoring phase. While in that phase, the kube-scheduler calculates a specific score for every one of those nodes, based on conditions like CpuPressure or MemoryPressure, energy efficiency indicators and results in a list of ranked nodes with a base score. The higher the value of a node score, the more likely this node will be finally, after more steps of the workflow, selected to host the pod from the scheduling queue.

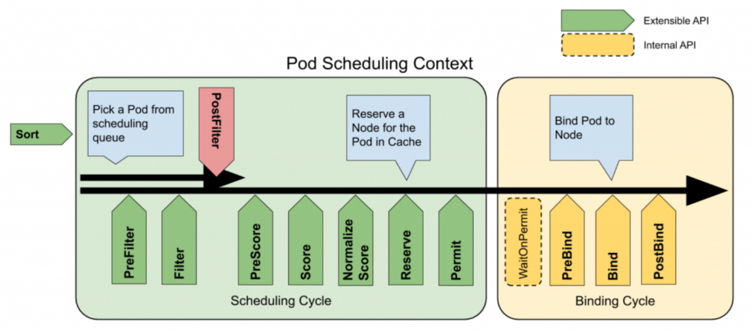


Figure 3.5: Kubernetes Scheduling workflow

(Source: Kubernetes documentation)

For extending any step of the scheduling workflow, the Kubernetes system provides an easy configuration field on the kube-scheduler manifest. With that, a single KubeSchedulerConfiguration can be passed an argument inside the kube-scheduler container. With this Kubernetes resource, the native scheduler procedure is prompted to communicate with a custom extender server before it concludes its scoring. This extender is declared inside the resource. The definition consists of its *urlPrefix*, meaning the ClusterIP service and port of the extender application pods, the verbs or the workflow steps which the kube-scheduler must be advised from the extender and finally a weight value which will multiply the node score list and will further affect the scheduling decisions. In the proposed framework, only the *prioritize* verb is significant for the Network-Aware Scheduling extension.

3.5.2: Latency Measurement

Before describing the workflow of the Scheduler Extender, it is important to clarify the supplementary applications and techniques required for this task. In order for the Extender to be able to receive latency measurements from a set of worker nodes, a *Latency-monitoring* application must be developed. A single instance of this application can be containerized and then be deployed distributed to the cluster with the use of a workload resource. Because the task necessitates the existence of exactly one *latency-monitoring* pod per worker node, the DaemonSet resource of Kubernetes was utilized. DaemonSet is deployed and keeps exactly one pod per node per application specified in the YAML file.

The application developed for use in the DaemonSet is an HTTP server with the Flask web microframework. It exposes many endpoints, but the main one benefiting the proposed framework is the */ping_client* endpoint. In this application, the *Ping3* Python module is used to provide an easy way to measure the Round Trip Time (RTT) in real-time and on demand of the Scheduler Extender. The ping command and therefore the */ping_client* endpoint, provide a set of mandatory or optional arguments for the execution, and all of them are posted as url parameters from the Scheduler Extender HTTP request. The optional parameters all have default values but for convenience, the time unit is selected to be milliseconds (ms), the timeout of the ping execution is proportional to the total size of the ICMP packet and finally the payload size which can vary form the default 56 to approximately 65.500 bytes. The ICMP packet total size is 8 bytes for the header plus the payload mentioned previously. In the context of the proposed system, which has principally to deal with video streaming via WebSocket connection, the selected payload size to be used on ping was 65.500 bytes, in order to achieve maximum ICMP packet size and therefore stress the system latency-wise. The only argument that must definitely be set is the IP of the desired client. The pod application then sends the traffic upwards to the interfaces layers, starting from the *veth*, to *flannel*, to *cni* and finally to the *eth1* interface from where it reaches the client. For the *latency-monitoring* application to have knowledge of the desired IP there is another feature to be implemented on the backend server application.

3.5.3: IP Acquiring

When an HTTP request from a client device, travels through the Ingress Controller and finally reaches a backend application checks its HTTP headers and looks for *X-Real-IP* to find the IP of the client, or, if absent, looks for the *X-Forwarded-For* header and specifically for the 1st IP on the x-forwarded list. If both of these headers are absent, then the fail-safe IP to acquire is a device’s static IP on the Testbed and test just the network capabilities of the worker nodes but not related to the specific client. After acquiring the IP information of the client, the backend server, while customizing the YAML template of the initializing Job, it adds an extra *client-ip* metadata label on the template of the Xpra-job. In that way, the Scheduler Extender will be able to also acquire this information and communicate it to the DaemonSet pods.

3.5.4: Scheduler Extender Server

This implemented Scheduler Extender is simpy an HTTP server developed over the Flask web microframework which exposes an endpoint for the *prioritize* verb of the kube-scheduler. The workflow of this Extender to initially have access to each of the filtered nodes provided to him. Utilizing the kubernetes API *in-cluster-configuration* in a similar manner to the backend application, the Extender inspects every pod in the cluster matching with the Label *“app: latency-monitoring”*, the application described above. From the pods it inspects, it communicates only with the ones residing on worker nodes which passed the filtering phase of the scheduling, the rest are bypassed.

In the extender’s phase when it tries to collect latency data, it loops through the available pods and requests their latency with the client from their */ping_client* endpoint with a suitable delay between the requests to avoid congestion issues. All latencies are then stored in a dictionary structure where the keys are the client’s IP matching to another dictionary value where the cluster node names are the keys and their respective values are their latencies with the client.

The final step for the Scheduler Extender is the score calculating phase. In this, for the same client which triggered the latency measurements, a score is calculated for every worker node which either communicated with or even for nodes that the ICMP request timed out. The score is anti-proportional with the client-worker latency, meaning that the higher the latency a link has, the lower the calculated score for scheduling.

```python
def calculate_scores(client_ip):
  client_dict = client_node_latency[client_ip]
  max_lat = max(client_dict.values())
  min_lat = min(client_dict.values())
  scores = dict(client_dict.items())
  for node, latency in client_dict.items():
    if latency:
      scores[node] = round(scale * (max_lat - latency)/(max_lat - min_lat)) +1
    else:
      scores[node] = 1
  return scores
```

In the snippet above is the function responsible for calculating scores for all the worker nodes which passed the filtering phase. Working on the dictionary entry of the specific client IP, the maximum and minimum latencies observed for this client are stored in their respective variables. For each node in the client’s dictionary, the relative percentage of this node’s latency to the range of observed latencies is calculated. The occurring value is on the scale from 0 to 1, so it is multiplied with a scaling factor, given that the lowest score in the kubernetes scheduling is 1. The scaled value is then rounded and an ace is added for the case of the maximum latency node, again to line with scheduling's minimum scoring. If a worker node's ping timed out, the latency observation is absent so this node gets the minimum scoring of 1. The scoring formula for an individual worker node is displayed below.

$$Score_{node} = \lceil Scale * \frac{latency_{max} - latency_{node}}{latency_{max} - latency_{min}} \rfloor + 1$$

Figure 3.6: Scoring formula for an individual worker node

Summarizing the proposed framework, when a client wants to run a virtual experiment, it sends an HTTP request to the backend service via the Ingress Controller NodePort service. The backend runtime acquires the X-Real-IP of the client device and assigns it on the Xpra-Job definition before requesting its creation from the Kubernetes API Server. Along the scheduling workflow, the Scoring phase sends an HTTP request to the Scheduler Extender for the */prioritize* verb. The Extender requests the observable latencies for client-worker links from all *latency-monitoring* pods of the DaemonSet, for filtered nodes only. After obtaining the observable latencies, the Extender calculates a score for each individual worker node, anti-proportional to the latency it provided. After constructing the node-score list it sends it back to the system's kube-scheduler to proceed with the scheduling workflow. The general architecture of the proposed framework after the addition of the Scheduler Extender is displayed below.

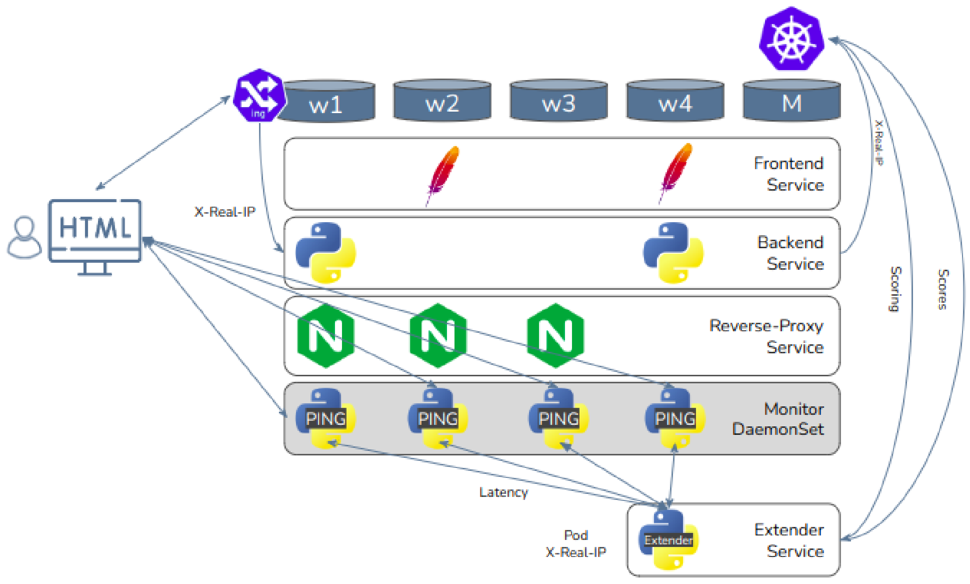


Figure 3.7: Latency-Aware Scheduling over LAN architecture

CHAPTER 4: EVALUATION

4.1: Overview

In this chapter, after developing and implementing the proposed framework in the available infrastructure of NITOS, it is evaluated to extract useful insights on the experimental value of the Latency-Aware Scheduling optimization. Initially, it presents a brief description on the additional implementation of metrics monitoring and measurement collection process as well as the evaluation setup. After that, an analytic presentation of the derived result is displayed through graph figures, while also discussing the implications of the derived results.

4.2: Evaluation Setup

4.2.1: Metrics and Tools

In order to evaluate the system, there should be objective metric indicators on the high level application of the proposed framework, the Xpra video streaming, which would reveal a variation between the native scheduling and the latency-aware scheduling. These metrics are provided by the Xpra utility, and more specifically on the Xpra server side inside the Pods. Off all the available metrics information, valuable for the context of the current evaluation are the frame rate of the video streaming and the quality of the image encoding of each frame. The former is found under *client.window.1.damage.fps*, or simply called Frames per Second (FPS) and has a range from 0 to 130 (maximum observed). The latter is located on *client.window.1.encoding.quality.cur*, or simply referred to as Quality, and has a standard range of 1 (minimum quality) to 100 (maximum quality).

Because the valuable information resides within an already functioning xpra-pod, the challenge of the evaluation was to access these metrics while running concurrently the video streaming task. This was achieved with the use of a sidecar container, the *xpra-monitor*, alongside the main Xpra container. The container image is a slim-buster debian distribution with only the Xpra utility and its dependencies installed. Because the *xpra-monitor* is defined in the same YAML template of the Xpra-jobs, three *emptyDirectory* volumes are mounted to both main-xpra and xpra-monitor, and it is configured for process namespace sharing. There are three directories linked and those are */run/xpra* where xpra-sockets files are stored, the user directory */run/user* and the Xpra root folder */root/.xpra*. Those three together link the two containers and let them function as if they were one host, so the one can run the server and the other take info for the same xpra instance.

4.2.2: Setup Architecture

The objective of this evaluation process is to test and prove that the system has actual promising effects on heterogeneous networks. As previously mentioned in Chapter 3: Proposed Framework, the worker nodes are selected based on resource homogeneity to reduce as much as possible their effect on the applications performance.

Another issue which demands addressing is the load balance between the nodes of interest. The cluster has many more applications to host which will have their own traffic handling requests and may cause congestion of traffic on their host node and influence the Xpra's auto-adjusting, especially when working as an Ingress controller or a Reverse Proxy server. Pods of all these applications are by default distributed among worker nodes with the inner algorithms and scoring of Kubernetes scheduling. Mendling mildly and legitimately with native scheduling, a *nodeSelector* attribute was added on the template's specification for the respective YAML definition of each cluster Deployment. In that way, all basic and core functionalities of the proposed system are hosted on a selected node, and for that same node, the Scheduler Extender is filtering it out, configured to ignore it from the scoring phase. The rest of the nodes are fully functional and hosting only the *latency-monitor* pods and are available for hosting exclusively Xpra-pods. The described setup architecture is utilized for the sole purpose of the evaluation and is not mandatory for the proposed framework to optimally function. It is displayed on the below diagram.

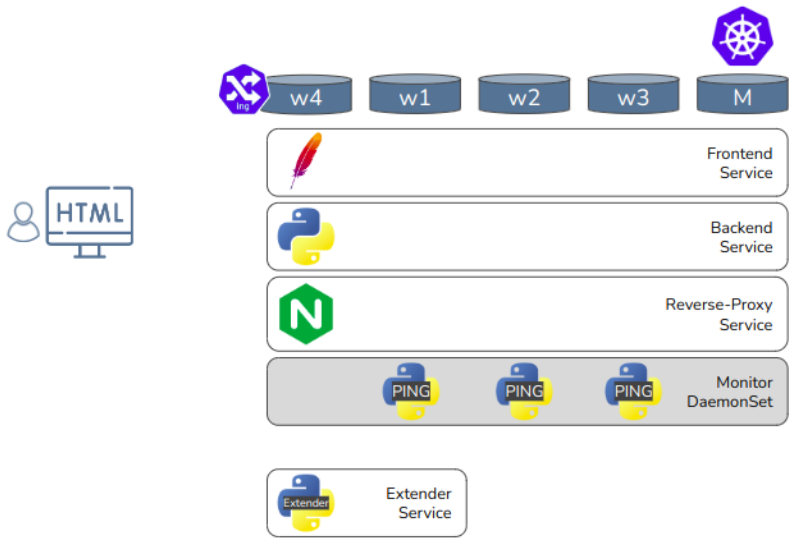


Figure 4.1: Evaluation phase setups architecture

Of the 5 given nodes of the cluster system, there is the master node M, the proxying node W4 and the rest of the nodes are tasked with simulating network heterogeneity within the cluster. Using Traffic Control a delay value of the millisecond class was added on each worker on their respective CNI interfaces, achieving link aggravation between them and both the client device and the rest of the cluster. In the above architecture displayed above the assigned latencies per worker are:

- Worker 1 (w1) => 0ms
- Worker 2 (w2) => 20ms
- Worker 3 (w3) => 50ms

4.2.3: Scenarios Description

The selected scenarios presented to demonstrate the effectiveness of the proposed framework, all tested with 6 virtual experiments, meaning 6 xpra instances, running per execution and all experiments had a 30 seconds duration before terminating. Not in the context of this thesis, previous evaluation results examining the scalability of hardware and network resources on the existing system, indicated a flat memory usage of roughly 350MiB per instance and a linearity of approximately 3 Mbit/s per instance. The total 18

Mbit/s of traffic is insignificant for the client's device networking because of the high tethered connectivity of the institutional network, reaching speeds of 1 Gbit/s. There will not be traffic congestion issues on the client part which could potentially affect Xpra auto-adjusting settings and reduce the metrics of interest.

In order to examine the improvement ratio and contribution of the proposed framework, there must first be an extensive understanding of Native Scheduling evaluation results. As previously mentioned in Chapter 2: Methodology and subsection 3.5.1 about Kubernetes Scheduling, it emphasizes aspects more related to hardware and energy efficiency. Because the Testbed nodes have relatively very good hardware aspects, the Native Scheduling tended to be partialed to specific nodes just because they had the capacity to host more pods. In order to take unbiased measurements of the system with Native Scheduling, an even instance distribution was preferred. Using the PodAntiAffinity attribute within Xpra-job YAML templates, the Native Scheduling was able to assign Xpra-pods in a Round Robin way. PodAntiAffinity with the hostname of each node just tries to avoid scheduling a pod of a certain application on the same node where another pod of this kind already exists. For the 6 instances per execution, an even distribution of 2 instances per node was achieved.

The Extended Scheduling of the proposed framework can be evaluated for multiple scenarios. According to the formulation of the scoring algorithm, the *Scale* factor can offer a great variety of possibilities for the instance distribution. The 1st scenario concerns a low scaling factor, valuing an absolute 10, and this results the scoring range to be between 1 and 10, and as consequence, the instance distribution is uneven, but all worker nodes take part in the workload share out. On the next step of the scaling scenarios, the factor of value 20 extends the maximum node score to 20 and so *w3* with maximum latency and minimum score was omitted from the scheduling procedure. Finally, the scenario where the scale factor, and thus latency, takes a bigger role, is with an absolute value of 50, the scheduling ends up being biased on the node with the best network link and that single node undertakes all instances. A descriptive graph of scenario distribution is shown below:

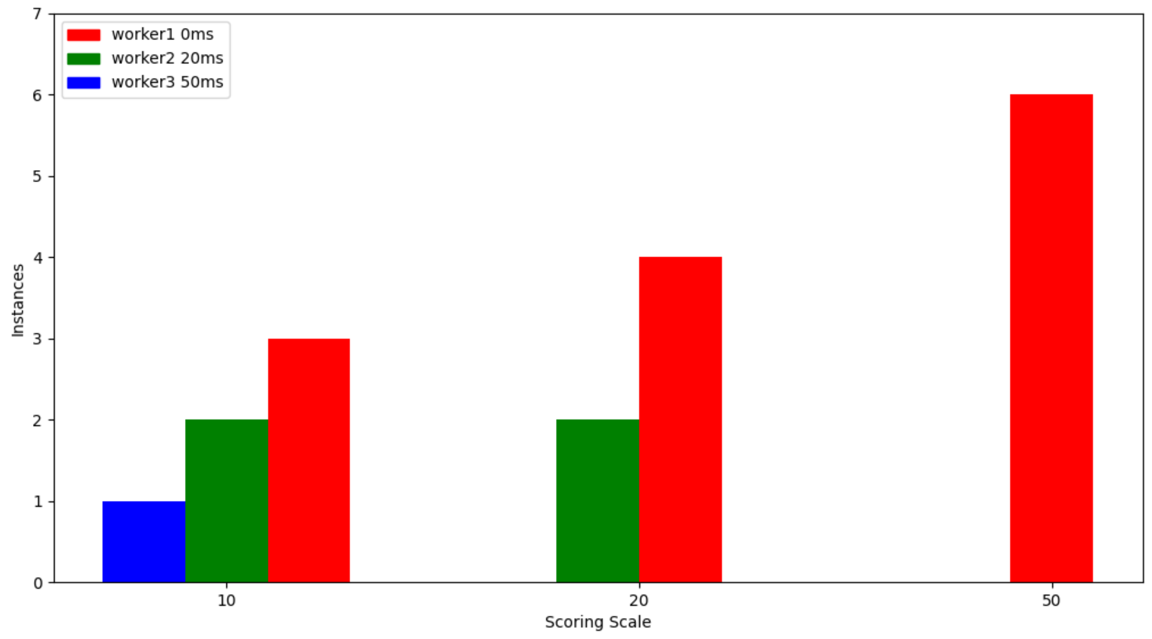


Figure 4.2: Xpra instances distribution on Extender Scheduling scenarios

4.3: Experiment Results

4.3.1: Native Scheduling

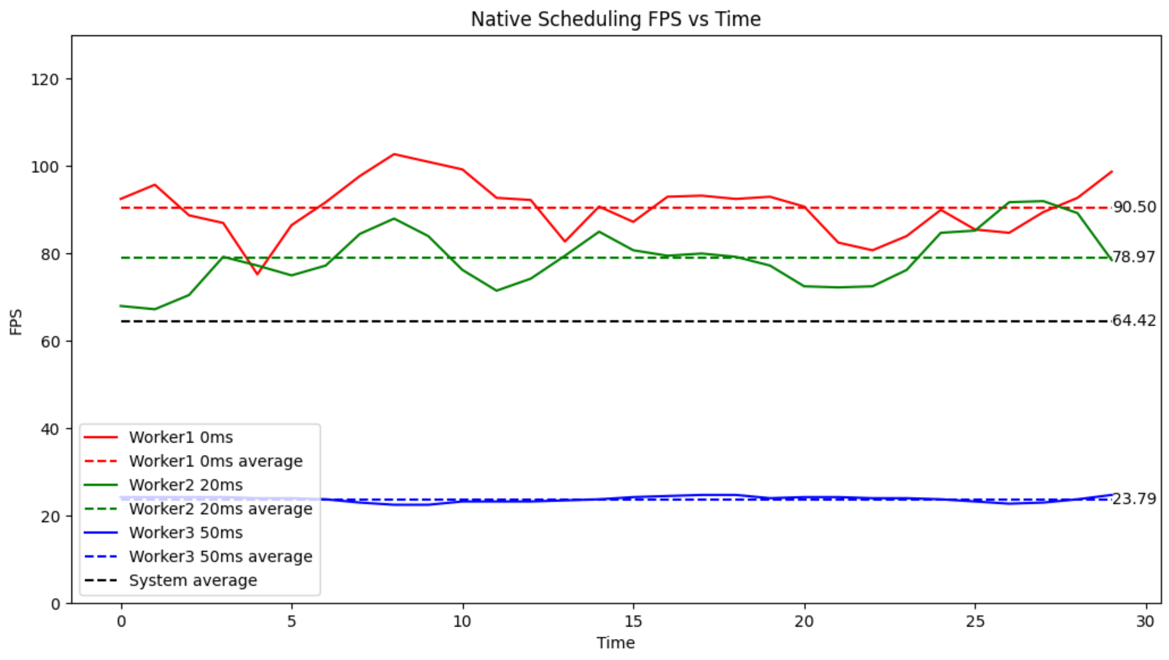


Figure 4.3: FPS vs Time for Native Scheduling scenario

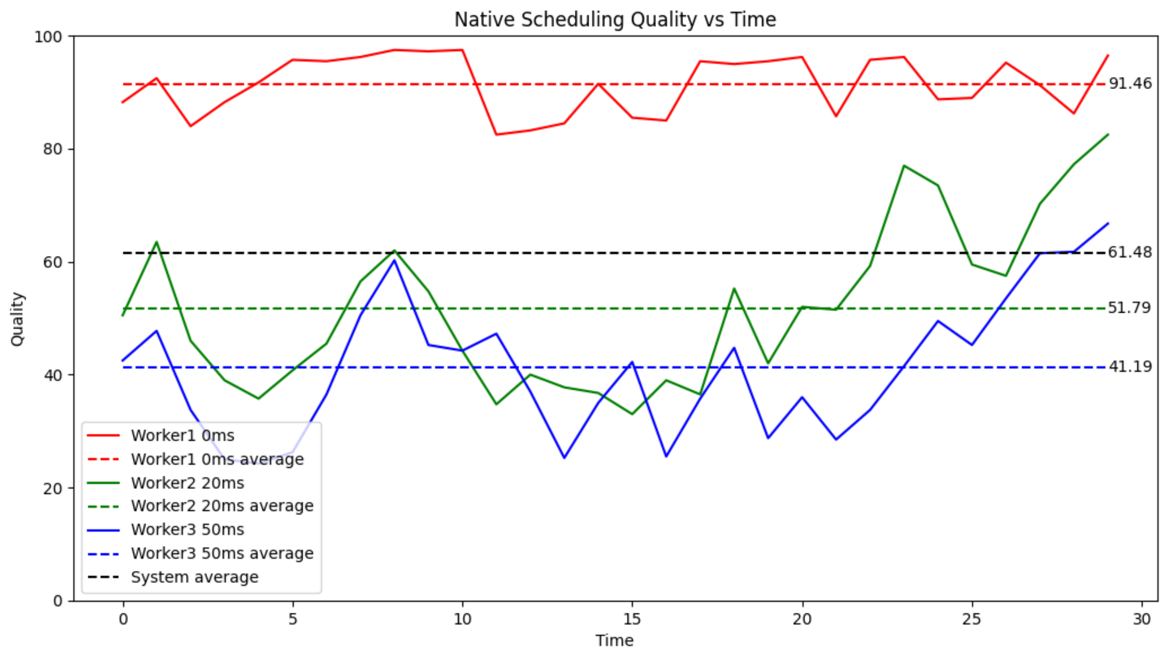


Figure 4.4: Quality vs Time for Native Scheduling scenario

As shown by the graphs above, each of the w1, w2 and w3 resulted in an average of 90.5, 78.97 and 23.79 FPS respectively, while they landed 91.46, 51.79 and 41.19 on the average Quality. From the total executions of this scenario, the system is on a base of 64.42 average FPS and 61.48 average Quality for all Xpra instances it serves.

4.3.2: Low Scoring Extended Scheduling

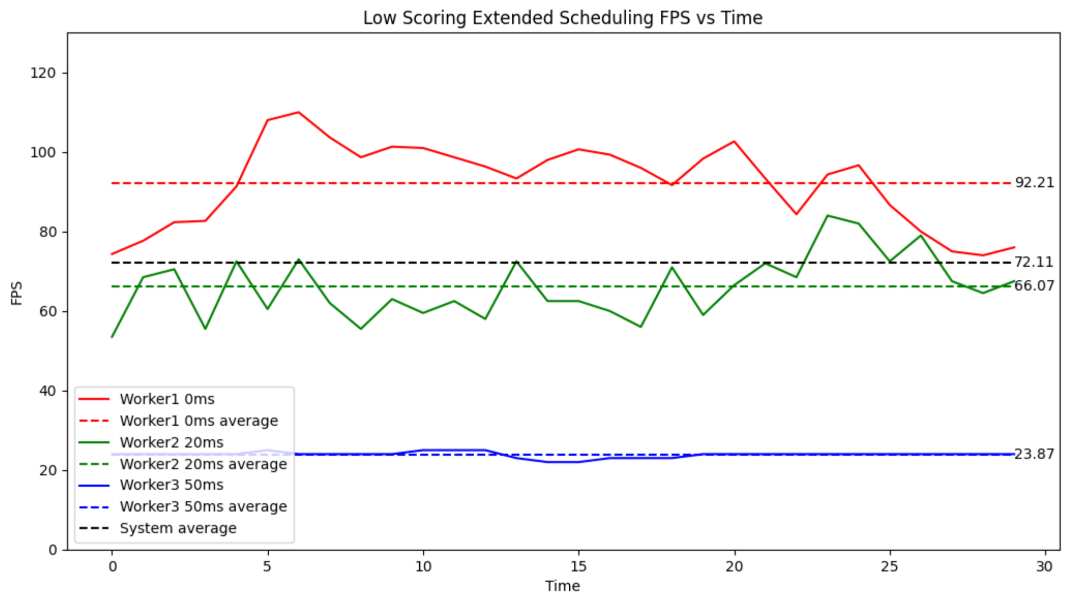


Figure 4.5: FPS vs Time for Low Scale Extended Scheduling scenario

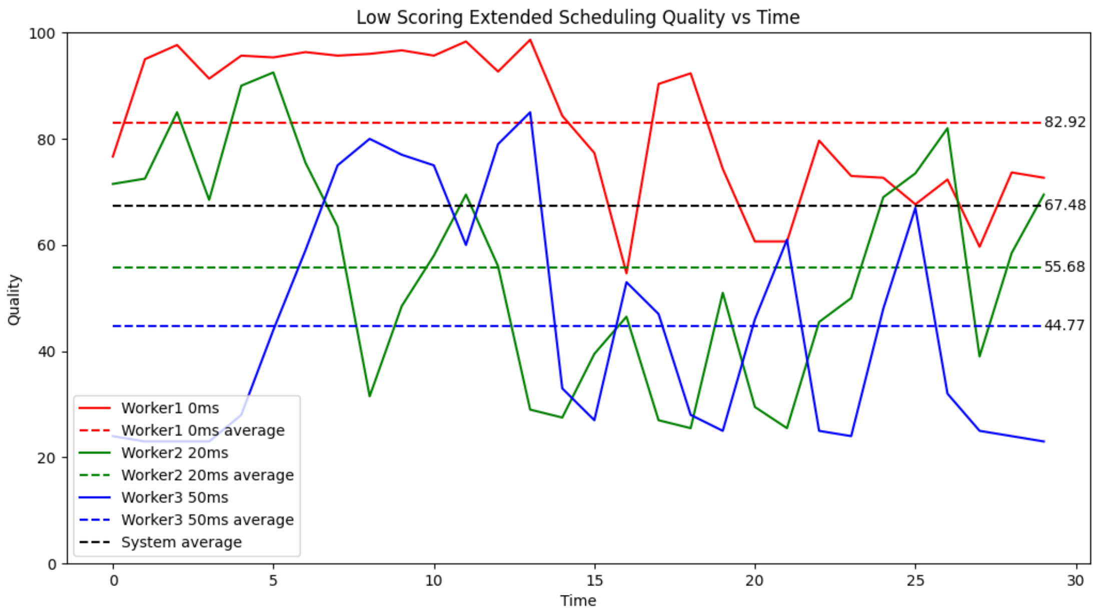


Figure 4.6: Quality vs Time for Low Scale Extended Scheduling scenario

As shown by the graphs above, each of the w1, w2 and w3 resulted in an average of 92.21, 66.07 and 23.87 FPS respectively, while they landed 82.92, 55.68 and 44.77 on the average Quality. From the total executions of this scenario, the proposed framework with a low scaling scoring achieves a 72.11 on average FPS and 67.48 on average Quality for all Xpra instances it serves.

4.3.3: Medium Scoring Extended Scheduling

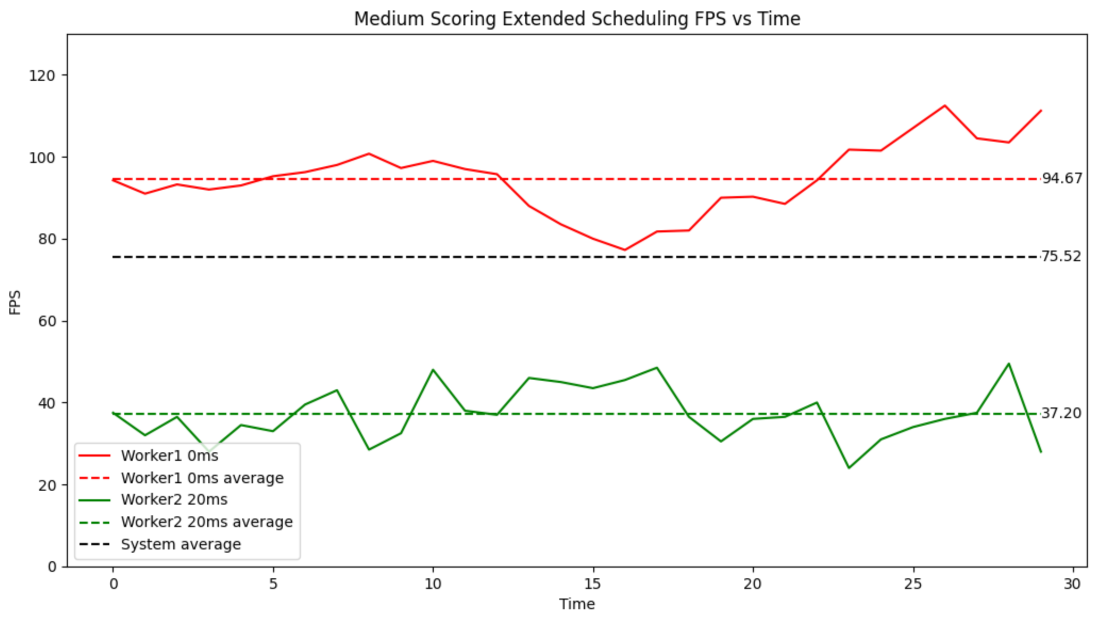


Figure 4.7: FPS vs Time for Medium Scale Extended Scheduling scenario

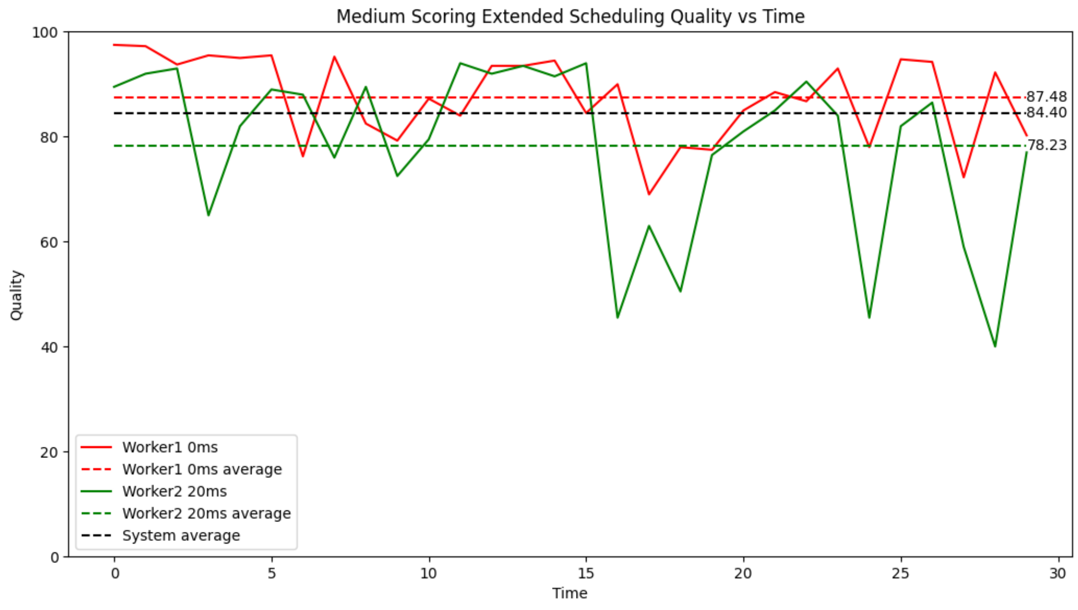


Figure 4.8: Quality vs Time for Medium Scale Extended Scheduling scenario

Now participating only in w1 and w2, the above graph displays an average of 94.67 and 37.20 FPS respectively, and the average Quality reaches 87.48 and 78.23. From the total executions of this scenario, the proposed framework with a medium scale scoring reaches 72.11 on average FPS and 67.48 on average Quality for all Xpra instances it serves.

4.3.4: High Scoring Extended Scheduling

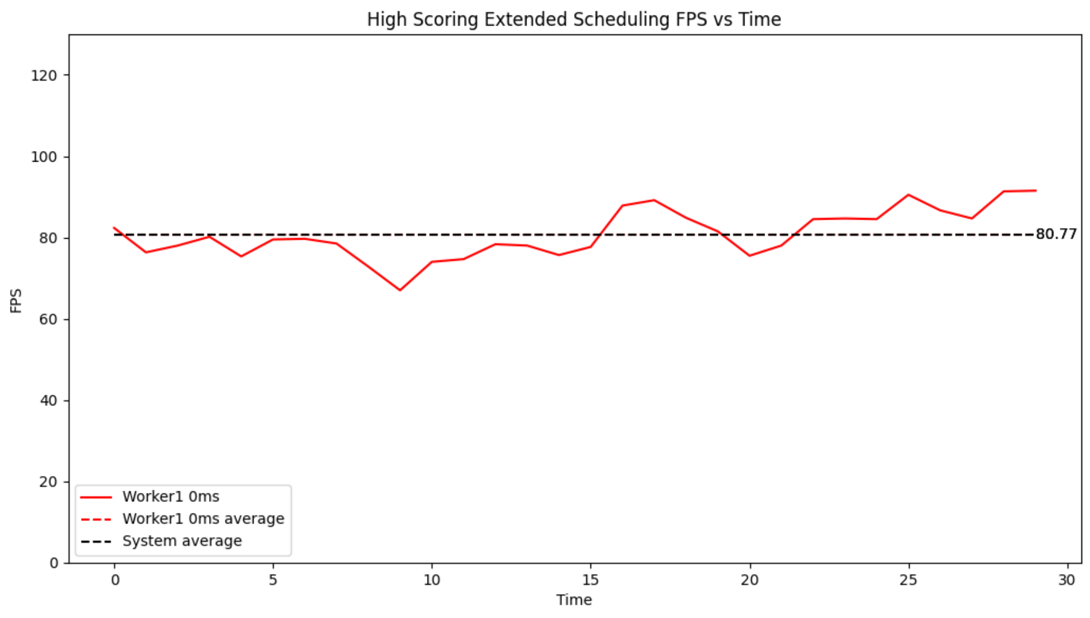


Figure 4.9: FPS vs Time for High Scale Extended Scheduling scenario

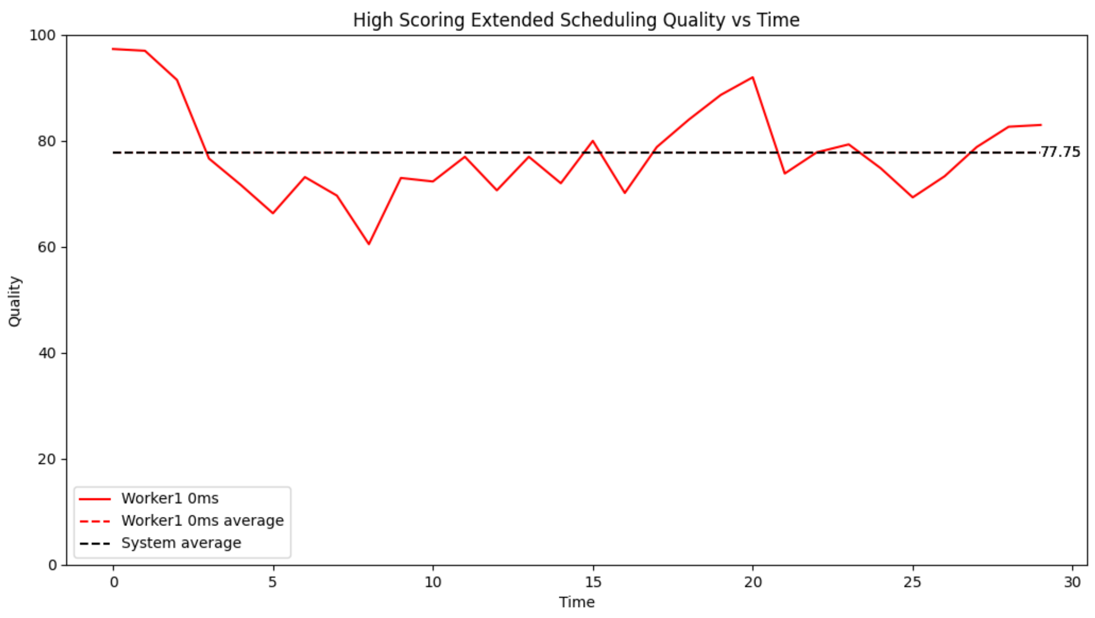


Figure 4.10: Quality vs Time for High Scale Extended Scheduling scenario

On the highest tested evaluation scenario with only w1 undertaking instances, all traffic passes through him. The above graph shows 80.77 on average FPS, and the average

Quality is 77.75. These numbers on absolute value are lower than the results previously shown from w1 with the minimum latency, but the proposed framework system overall presents an increase.

4.3.5: Evaluation Results

Below are displayed the collected values for average FPS and average Quality from the totality of evaluation scenarios and experiments.

Scheduling/Metric	**FPS**	**Quality**
Native	64.42	61.48
Low Scale Extended	72.11	67.48
Medium Scale Extended	75.52	84.40
High Scale Extended	80.77	77.75

Table 1.1: Overall System outputs for Native and Extended Scheduling

The above measurements show a promising evaluation for the proposed framework because it displays a relative increase on every single scenario for both FPS and Quality metrics. The interesting thing to discuss from the results is the peak on average Quality with a medium scale scoring while the average FPS for the same scenario remains lower than the high scale scenario. This is a drawback for CPU intensive work a single node has to make in the high scale scenario which results in a least optimal solution for Quality but still the most effective for FPS increase.

Scheduling/Metric	**FPS**	**Quality**
Low Scale Extended	11.9%	9.8%
Medium Scale Extended	17.2%	37.3%
High Scale Extended	25.4%	26.5%

Table 1.2: Extended Scheduling evaluation of the Proposed Framework

CHAPTER 5: CONCLUSION

5.1: Outputs and Contribution

Summarizing this master thesis, a full implementation of an orchestrated cluster system was presented. The existing RET system was successfully transitioned to a distributed containerized system and all challenges regarding the implementation within the Testbed infrastructure were addressed. The developed system is not only able to serve containerized GUI applications to clients on LAN but also achieves this in a cloud oriented manner. The proposed framework of the current thesis is completed with the addition of Latency-Aware extension for the native Kubernetes scheduling workflow, targeting the general improvement of Quality of Service attributes, which was achieved. The distributed Remote Desktop Control applications of interest, are scheduled to be hosted on nodes based on a latency related prioritization, acquiring the best possible network link between worker and client. The proposed framework and scoring algorithm is evaluated resulting in positive and promising improvements on Quality of Service.

5.2: Future Work

Various techniques and concepts offer an interesting view towards improving and extending the proposed framework of the current work. A major update can be the development of a recurring latency monitoring daemon which is constantly updated for client-worker network links and works in a stateful approach, adding or deleting recent clients on its runtime memory. In that context, it would be very interesting to implement a rescheduling mechanism from either on demand latency measurements or the mentioned recurring latency monitoring. This would allow the adaptability of the system in case a worker node suffers from a failure or network deprivation, or in case a client changes his network conditions. If experimentation with a real cloud provider, many reconfigurations and adjustments should be done to the system according to the cloud infrastructure. This could potentially pave the road to substitute current ClusterIP services with LoadBalancer services and work on load balancing algorithms and mechanisms, rather than or alongside scheduling extensions.

References

J. Santos, T. Wauters, B. Volckaert and F. De Turck, "Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing Applications," 2019 IEEE Conference on Network Softwarization (NetSoft), Paris, France, 2019, pp. 351-359, doi: 10.1109/NETSOFT.2019.8806671.

A. Marchese and O. Tomarchio, "Network-Aware Container Placement in Cloud-Edge Kubernetes Clusters," 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Taormina, Italy, 2022, pp. 859-865, doi: 10.1109/CCGrid54584.2022.00102.