FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Cloud Tools for Crowdsourced Live Journalism

Rui André Rebolo Fernandes Leixo

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Maria Teresa Magalhães da Silva Pinto de Andrade

Co-supervisor at the receiving entity: Vasco Manuel Pérola Filipe

July 22, 2020

Cloud Tools for Crowdsourced Live Journalism

Rui André Rebolo Fernandes Leixo

Mestrado Integrado em Engenharia Informática e Computação

July 22, 2020

Abstract

The democratisation of record ready mobile devices, capable of audio and video transmission, coupled with pervasive access to an internet connection, have contributed to an increase in Crowdsourced Live Journalism. Crowdsourced Live Journalism consists of amateur live video feed recorded by the general public, used by a news company, thus providing ample video coverage to many times uncovered situations. These live contents are usually destined to a mainstream social network platform. Mainstream social networks usually rest on a centralised architecture, that is, a data centre owned, operated and maintained by a given social network company. This originates trust issues not only concerning user privacy (since there is no guarantee user data is properly handled and not intentionally misused by the company) but also on trusting the content created and shared on the platform, existing no way to assess contents creation and modification.

Another problem which arises from centralised cloud nature, is the high cost to attain good user experience in large venues (sports stadiums, conference centres, concert halls). This happens due to significant delays and limited available bandwidth when communicating with the central server. Improving the current infrastructure by simply adding more resources would result in an over costly solution.

Both of the aforementioned problems could be tackled by developing a decentralised Crowdsourced Live platform with an edge-cloud network as the underlying infrastructure, formed by an interconnection of small servers placed near the end-user. This would contribute to the reduction of the delay experienced by the user on content retrieval and creation, whilst also providing the necessary flexibility to adapt to the platform's growing needs.

In this work the related literature was analysed, particularly the fields of Cloud/Edge Computing, virtualisation, monitoring, media and networking by the virtue of their contribution to the project's realisation.

Implementation followed, where a scalable Edge Cloud architecture was implemented, using Kubernetes as the orchestration engine. It is capable of scaling accordingly to the platform's current needs and withstand some components failure, automatically recovering them. Furthermore, an ad-hoc stream fallback recovery mechanism was developed, enabling a stream's complete recovery in the event of its transcoder malfunction.

Lastly, the solution was validated in a real-world like scenario, attesting its efficacy for a real-time live streaming platform. Better end to end latency was verified when compared to a traditional central cloud approach. Along side with lowering central cloud's bandwidth consumption.

Keywords: crowdsourcing, cloud computing, edge computing, edge architecture

Resumo

A disseminação de dispositivos móveis equipados com câmaras, capazes de difundir áudio e vídeo, em conjugação com uma constante ligação à *internet*, têm contribuído para o aumento do Jornalismo Colaborativo ao Vivo. O Jornalismo Colaborativo ao Vivo, consiste na gravação de vídeo amador em direto, usado por uma empresa noticiosa. Isto permite a obtenção de vastos conteúdos em vídeo, que incidem sobre situações que de outra forma ficariam sem cobertura. Estes conteúdos têm muitas vezes como destino uma rede social convencional, normalmente assente sobre uma arquitetura centralizada. Ou seja, uma arquitetura na qual o *data centre* é mantido, controlado ou pertencente a uma empresa singular. Isto poderá originar problemas de confiança, não apenas no que toca a dados pessoais de um utilizador (visto que não há qualquer garantia de que estes são processados e armazenados de forma adequada), bem como no conteúdo criado e partilhado na plataforma, devido à falta de mecanismos para garantir a sua proveniência e historial de modificações.

Outro problema emergente de um ambiente de *cloud* centralizada, é o elevado custo de proporcionar uma boa experiência de utilização em grandes recintos (estádios, centros de conferências, salas de espetáculos). Isto ocorre devido a um atraso considerável e limite na largura de banda existentes nas comunicações com o servidor central. Neste caso, a tentativa de melhoria da infraestrutura, simplesmente adicionando mais recursos, resultaria numa solução deveras custosa.

Ambos os problemas acima referidos poderiam ser mitigados ao combinar uma plataforma de Jornalismo Colaborativo ao Vivo, com uma arquitetura de *edge-cloud* como infraestrutura subjacente, formada por pequenos servidores interconectados, próximos do utilizador final. Isto iria contribuir para a redução do atraso experienciado pelo utilizador na obtenção e criação de conteúdos, ao mesmo tempo permitindo a flexibilidade requerida para o crescimento da plataforma.

Neste trabalho, o estado de arte relevante foi analisado, particularmente os campos de *Cloud/ Edge Computing*, virtualização, monitorização, media e redes, pela sua determinante importância para a elaboração do projeto. Seguiu-se a implementação, onde uma arquitetura de *Edge Cloud* escalável foi concebida, usando o Kubernetes como "orquestrador". Esta é capaz de escalar mediante o estado da plataforma e de lidar com a falha de alguns componentes, recuperando-os automaticamente. Ademais, um mecanismo *ad-hoc* de recuperação da transmissão foi desenvolvido, permitindo que esta recupere completamente caso haja uma falha com seu transcodificador.

Por último, a solução desenvolvida foi validada aproximando-se de um ambiente real, atestando a sua eficácia para uma plataforma de transmissão em tempo real. Verificou-se um menor atraso ponta a ponta na transmissão comparativamente com uma solução mais tradicional de *cloud* centralizada. Ao mesmo tempo, registou-se um menor consumo de largura de banda na *cloud* central.

Keywords: crowdsourcing, computação na nuvem, computação edge, arquitetura edge

Acknowledgements

Firstly, I solemnly express my deepest gratitude to anyone who supported the writing of this dissertation.

Following, I thank my family and friends who were always there for me with no hesitation nor reserves.

I would like to thank my faculty colleagues, with whom learnt a lot and without whom this five years couldn't possibly have been the same.

To Professor Maria Teresa Andrade, my supervisor, for her investigation and academic expertise which were huge contributes to this dissertation improvement.

Furthermore, I would like MOG Technologies for the challenge proposed and the fantastic working conditions provided. Additionally, I would like to especially thank my colleagues at MOG Academy with whom I worked during this semester.

Finally, I would like to express my appreciation to Alexandre Ulisses, Ivone Amorim and Vasco Filipe, whose contributions and insight were paramount for this work's success.

Rui Leixo

"You must put your head into the lion's mouth
if the performance is to be a success"

Winston Churchill

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
CLI	Command Line Interface
CNCF	Cloud Native Computing Foundation
CNI	Container Network Interface
CNAME	Canonical Name
CPU	Central Processing Unit
DNS	Domain Name Server
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
I/O	Input/Output
IP	Internet Protocol
JSON	JavaScript Object Notation
LAN	Local Area Network
LXC	Linux Containers
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NIST	National Institute of Standards and Technology
OS	Operating System
OSI	Open System Interconnection
PaaS	Platform as a Service
QoE	Quality of Experience
QoS	Quality of Service
RAM	Random Access Memory
RTMP	Realtime Messaging Protocol
RTP	Realtime Transport Protocol
SaaS	Software as a Service
SQL	Structured Query Language
TCP	Transmission Control Protocol
TSDB	Time Series Database
UDP	User Datagram Protocol
VM	Virtual Machine
VMM	Virtual Machine Monitor
VXLAN	Virtual Extensible LAN

Chapter 1

Introduction

During the last decade, mobile devices have seen widespread adoption, which has contributed to an increase in video traffic on the Internet. Cisco's 2020 Global Networking Trends predicts video traffic will represent 82% of all business network traffic by 2022 [8]. Cisco also foresees that 70% of global population will have at least one mobile device by 2023 [7]. Such devices are equipped with high definition camera sensors, capable of capturing what's before its user with great fidelity.

Crowdsourced video streaming has been fostered by the ubiquitous existence of mobile devices with sufficient internet connectivity. It can be seen as a shift on the content creation paradigm, where regular users with consumer grade equipment stream content to the former Content Provider (CP) (eg. news outlet, VOD provider, social network platform) [4]. Furthermore, live streams often need to be transcoded (reencoded in another encoding or bitrate) so they have the correct encoding, that is, appropriate for the final media platform.

Cloud computing may seem a good paradigm to employ to realtime video encoding, however it is usually located many hops away, far from the end-user. Realtime applications for instance: video livestreaming, online gaming, autonomous driving, require a stable, high bandwidth, and low latency internet connection, from client to server. Failing to meet said requirements, can introduce significant delay, and interfere with its realtime nature, therefore leading to Quality of Experience (QoE) degradation.

Live streams are commonly destined to mainstream social network platforms. Mainstream social networks usually rest on a centralised cloud architecture [14], commonly owned and managed by a single entity, the social network company. This poses serious trust issues concerning user privacy, as there is no guaranty user data is properly handled. Moreover, there is no way to assess content created or shared on the platform, neither its creation nor modification.

Both of the aforementioned problems could be tackled by developing a decentralised Crowdsourced Live platform with an edge-cloud network as the underlying infrastructure, formed by an interconnection of small servers placed near the end user. This would contribute to the reduction of the delay experienced by the user on content retrieval and creation, whilst also providing the necessary flexibility to adapt to the platform's growing needs.

1.1 Context

This work focuses on investigating approaches and proposing a solution to combine the use of centralised cloud infrastructures with the edge cloud paradigm, to overcome the above identified problems. This was carried out in a corporate environment at MOG Technologies in Maia, Porto. MOG Technologies is a company inserted in the broadcasting and professional multimedia industry.

1.2 Motivation

The use of an Edge Cloud in large public venues would contribute to a better experience to end-users of Crowdsourced Live platforms, reducing latency on content upload, at a lower cost than what could be achieved by solely using a centralised solution, or a permanent solution on the event site.

1.3 Objectives

The main goal of this dissertation is to devise and implement an architecture combining centralised cloud infrastructure with benefits of the edge cloud paradigm to support a crowdsourced live streaming platform with the use case of live journalism. This architecture aims to be able to better cope with the increase in client demand in large venues, scaling according to the social platform's needs. Lowering infrastructure costs is also expected when compared to permanent, non scalable infrastructure on venue. The prototype is to be subjected to real-world like validation.

1.4 Document Structure

Apart from Chapter 1 this document has 4 more chapters.

Chapter 2 is the State of the Art, focusing on Cloud, Edge Computing, Virtualisation, Orchestration, Monitoring, Media and Networking.

Chapter 3 discusses a solution to the problem at hand, presenting and explaining its components.

Chapter 4 details the validation environment, test list, our expectations and discusses the verified results.

Finally, in Chapter 5 conclusions are drawn regarding the whole document.

Chapter 2

State of the Art

This chapter focuses on key technologies and solutions found during the literature review. Topics relevant to the creation of the architecture are discussed, namely: cloud computing, edge computing, virtualisation technologies, orchestration and monitoring.

Firstly, cloud and edge computing are reviewed as they are the cornerstone of the project. Secondly, virtualisation technologies are addressed due to the fact they provide isolation of provisioned resources, allowing for cloud scalability as well as resource monitoring. Thirdly, orchestration and monitoring are discussed by virtue of their ability to coordinate the different services and assess their performance.

Finally, crowdsourcing, micro service architecture, media and networking topics are discussed because of their pertinence in this project's implementation, covering its many facets .

2.1 Cloud Computing

Cloud Computing is a paradigm that provides ubiquitous, convenient, on-demand computing resources, allowing for its rapid provisioning or release with minimal service provider's required interaction [20].

The National Institute of Standards and Technology (NIST) defined a model that serves as a baseline for cloud comparison and discussion, it is the defacto reference model [1].

NIST's Cloud model has the following five essential characteristics:

- **On-demand self-service** A consumer can unilaterally request for computing capabilities, such as server time and storage. This happens without the need for human interaction on the service provider's end.
- **Broad Network Access** Computing resources are accessible over the network through standard practices from a wide range of heterogeneous platforms.

[1]NIST's statement on cloud computing definition - `https://www.nist.gov/news-events/news/2011/10/final-version-nist-cloud-computing-definition-published` [Last access, 4 February 2020]

- **Resource Pooling** The provider resources are pooled to serve multiple tenants simultaneously, fulfilling their needs independently. The consumer has no control over the exact location of the provided resources.
- **Rapid Elasticity** The model scales dynamically depending on the consumer's current demands. To the consumer appears as if the Cloud has unlimited resources.
- **Measured Service** Resource usage can be logged, providing a report to the consumer containing metrics on all the measured resources.

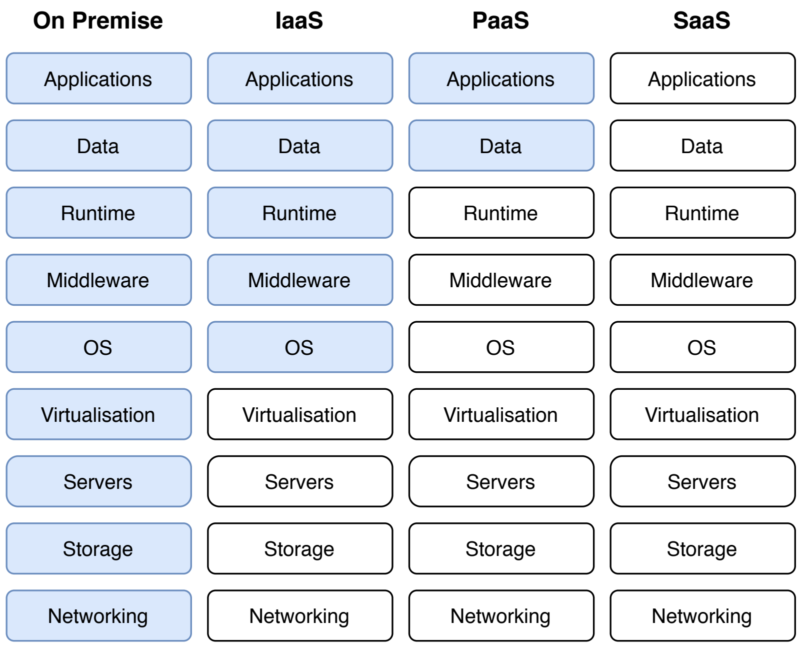

Figure 2.1: Cloud computing service models

Cloud Computing can fall into one of these service models [20], each of them having increasingly more user (consumer) abstraction:

- **Infrastructure as a Service (IaaS)** Fundamental computing resources are provided to the consumer, namely processing, storage and networking. The consumer is able to deploy and run arbitrary software, from operating systems to applications. However, the underlying cloud infrastructure is abstracted from the consumer by the provider, having the former no control over it.
- **Platform as a Service (PaaS)** In this category, the consumer has control over the application running on the platform. The platform encompasses the OS, libraries and the cloud infrastructure.
- **Software as a Service (SaaS)** The consumer here does not manage any of the existing layers, getting only to use the provider's applications running on the cloud infrastructure.

This is depicted in Figure 2.1, where the layers in blue are managed by the consumer and the clear ones by the service provider.

2.1.1 Deployment Models

A private cloud is used exclusively by a single organisation and may be located on or off-premises. Private clouds provide more precise control of the infrastructure, although many times being a more costly solution. They can be operated by the owning organisation, or by a third-party company.

A community cloud is jointly used by a specific community of consumers, from companies that share a common goal or policy. It may be owned, managed and operated by one or more community organisations or by a third-party. Its location may be on or off-premises.

A public cloud is intended to be used by the general public. It may be owned, managed and operated by a business, academic or governmental organisation. It exists on premises of the cloud provider.

A hybrid cloud infrastructure can be defined as a composition of two or more distinct cloud infrastructures, that remain unique identities but are bound by a communication layer enabling data sharing.

2.1.2 Edge Computing

Edge Computing, is a paradigm where computation is performed at the edge of the network. It can be defined as the presence of any computing and network resources located between data sources and cloud data centres [24].

The Edge is more efficient than the cloud in some computational tasks, where data is produced at the edge of the network, as it tries to shorten the introduced delay in communications from user devices to the central cloud infrastructure [26]. The reduced latency, high bandwidth and low jitter results in better user experience in applications that have real-time requirements or whose end-users produce a large volume of data. Although it lacks the processing power of the cloud, data processed at the edge would ease the central cloud, translating to a shorter response time and lowering bandwidth consumed to the central cloud .

Considering the heterogeneity of consumer media devices, which support different formats and possess different networking and computational restrictions, in order to the provide the best QoE while keeping costs and bandwidth consumption at a minimum, media has been transcoded at the edge before being sent to consumers. This way the Content Provider can possess only the source version of the requested video, that will then be transcoded dynamically at the edge to a suitable encoding for the viewer. Furthermore, the reverse also happens, as media production has shifted, a crowd-source — a regular user with a handheld device and sufficient Internet connectivity that produces media — can live stream a considerable amount of data to the content provider, afterwards to be transmitted to viewers [3]. In both these scenarios, media can be processed at the edge of the network, thus reducing bandwidth consumption and leveraging the proximity to the consumer or producer.

2.1.3 Comparison

Cloud computing possesses seemingly infinite resources which can be rented on a per user basis, depending on the user's current needs making it formidable for high intensity and short lived workflows, as the user can benefit from its huge computational power while not having to worry with colossal expenses such an permanent infrastructure would entail.

Despite all its virtues, Cloud Computing is ill suited for realtime tasks, as the data centre is many times far away from the consumer, what's linked to considerable networking delay and low bandwidth. Edge Computing appears and new alternative to a completely Central Cloud approach, were less powerful servers are place a the edge of the consumer's network, therefore enjoying LAN proprieties, low latency and bandwidth. Tasks can be preprocessed at the edge, what's called cloud offloading and only the resulting data, relayed to the central data-centre.

2.2 Virtualisation

Virtualisation technologies are used in Cloud Computing, as they allow the isolation of provisioned resources, cloud scalability and resource monitoring. Their use leads to lower resource waste, as a *host* system shares its hardware resources with all the *guest* OSs.

These technologies are essential in a cloud environment by virtue of their ability to provide fault isolation [25] — if a component fails it will not jeopardise others — as well as resource isolation and cloud scalability. Some of the existing virtualisation models are discussed on the following subsections.

2.2.1 Virtual Machines

Virtual Machines virtualise the whole stack from hardware to software. This allows great flexibility when there is the need to emulate a machine using a different OS or a distinct hardware architecture [22]. VMs are easy to replicate (in case of failure, or to attain redundancy), easy to backup and to migrate to a different location. With VMs it is possible to replicate the functions of multiple physical servers on a single physical machine.

However, there are some drawbacks to VM use like performance overhead and security issues. The performance overhead is present when compared to running the application on a non virtualised physical machine thus translating to an increase in execution time [25].

The Hypervisor or Virtual Machine Monitor is the main software that handles the creation and execution of virtualised instances. It allocates the physical resources to the virtual instances and provides them, a similar interface to the one present when they run on a real system. The Hypervisor also has to have security guarantees, system isolation must be upheld, that is, a virtual machine application shouldn't be able to interact with other machines applications and resources [25].

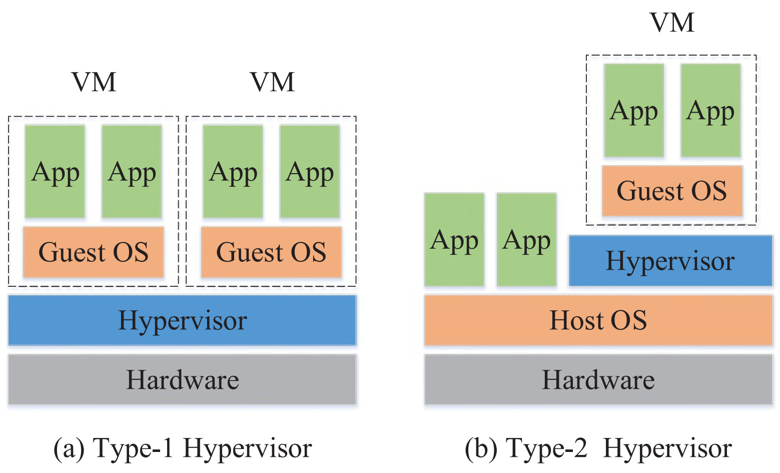

Figure 2.2: Hypervisor types [25]

There are two types of Hypervisors, bare-metal (type-1) and hosted (typed-2) as shown in Figure 2.2. Type-1 runs directly on top of the physical machine's hardware, on the other hand a Type-2 Hypervisor rests on top of a conventional OS, what sometimes implies the provided guest API is implemented by the means of the host's own interface [25].

Some widely used Virtual Machine Monitors are: Xen (type-1), VMware Workstation (type-2), KVM (type1 and type2), Virtualbox (type-2).

2.2.2 Container based

Containers are a lightweight alternative to hypervisor-based virtualisation, the latter abstracts hardware, virtual device drivers and typically runs a full OS. In contrast, containers emulate only the user space, as they are run on top of the host's operating system kernel [22, 27]. The kernel and OS libraries are shared between all the existing containers. Consequently, containers can achieve higher density on a single host, as well as smaller disk images when compared to VMs [19]. In addition, containers are extremely fast to instantiate and can also be quickly paused and resumed. Conversely, containers also have some disadvantages, due to the shared kernel approach only an OS that shares the same kernel as the host can be virtualised, i.e. a Linux host can't run Windows containers. Also, some of the isolation is lost, as the host kernel is exposed to the containers, which can lead to a multi-tenant security problem [22, 19].

2.2.2.1 Docker

Docker [9] is an open-source container implementation that incorporates Linux namespaces and control groups (cgroups) similarly to LXC (see Figure 2.3). It attempts to solve problems, affecting development and deployment of many modern applications: multiple component applications, each module's own set of dependencies, inter-module dependency incompatibilities [21].

A docker container is an isolated environment created from a docker image, a blueprint that specifies the required dependencies, configurations and applications pertaining to said environment.

Contributing to its portability, docker images can be pushed (uploaded) and pulled (downloaded) from a registry. Images can be based on other docker images, sparing the need to redo images from the ground up when only slight adjustments are required.

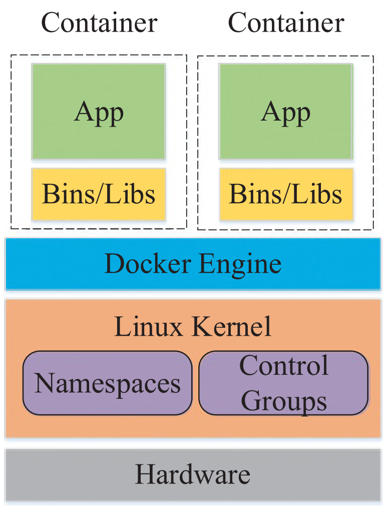

Figure 2.3: Architectural overview of the Docker container [25]

2.2.2.2 Docker Networks

Containers can communicate with each other, in a similar fashion to what virtual machines do, via virtual networks. A container network can have one of the following types [10], translating to different levels of isolation:

- None - Disables container networking, only keeps the Loopback inteface
- Bridge - This is the default network driver. A bridge network connects every container belonging to the network as if the were in a LAN. They are isolated from the host machine.
- Host - Removes network isolation by connecting the containers to the host network. Only works on Linux hosts.
- Overlay - Provides a network that can connect multiple Docker daemons, thus enabling containers on different hosts to be easily connect for Swarm communication. Furthermore no OS level routing is required to maintain an overlay network
- MacVlan - Assigns a MAC address to the container, treating it as a regular physical device. Suitable for legacy applications which expect connection to a real physical network.
- Third-party network plugins - Non officially maintained network plugins, like CNIs (refer to 2.5)

2.2.3 Unikernel

Unikernel, sometimes referred to as LibraryOS, takes a different approach at virtualisation, as it aims to be more secure and light, only having a slimmed down version of the a traditional OS kernel. This kernel only includes the bare minimum to allow the core application execution, the needed system calls and services. The minimalist take on the kernel results in a diminishing of the attack surface, making it more secure [25]. The vast majority of Unikernel approaches don't waste CPU cycles emulating the timer interrupt unlike hypervisor which forgo energy doing nothing [5].

Manco et al. [19] introduce LightVM, a middle ground between hypervisors and containers, a redesign of Xen's toolstack optimised for performance, that can boot a minimalistic VM (unikernel) in 2.3 ms. It has also almost constant creation and boot times, regardless the number of running instances. In spite of its remarkable performance, it is not as easy to use (requiring more development effort) and does not have the large tools ecosystem containers possess.

Braterrud et al. [5] present IncludeOS a minimal resource usage, single tasking unikernel, providing developers a novel way to build C++ code directly into a virtual machine at compile-time. It is easy to develop, only requiring a single include. When compiled, the build system selects solely the needed services originating a single binary, which can then be booted with the addition of a boot sector. IncludeOS can be virtualised using any x86 virtualisation environment.

The authors developed a simple DNS server that required 158 KB of persistent memory and required 5 to 20% less CPU time, when compared to the same binary running on Linux.

2.2.4 Comparison

Virtual Machines emulate the complete hardware and software stack, what makes them highly versatile as they are able to virtualise different hardware and their OS's irrespective of the host's hardware architecture. They also possess a high degree of isolation, what makes VMs very secure. However, these qualities also make Hypervisor based approaches, slower to start and more resource intensive than the other two presented alternatives, hence not as suitable for a cloud scenario.

Containers, many times called lightweight VMs, on the other hand only emulate the user space, sharing the hosts kernel across every container. As a consequence, they are lighter than conventional VMs, quicker to boot but are limited to virtualise operating systems compatible with the host's kernel.

Lastly, Unikernels, take a minimalist approach to virtualisation. They only have what is truly crucial for the task they were designed for. They don't even have a complete kernel as they only really include the bare minimum system calls to perform their mission. This reduced attack surface makes them more secure as there is less code which can exploited. Conversely, their benefits makes them rather rigid, as they aren't as easy to develop for.

2.3 Orchestration

Managing a plethora of containers, many times scattered among distant and heterogeneous systems, is a very complex task. Orchestrators provide a solution for this problem, coordinating at software and hardware level the deployment of a set of virtualised services, in order to fulfil certain operational and quality objectives [27].

Service Level Agreements describe a fixed set of guaranteed states of the networks topology. States are validated based on several metrics, namely: user requests, application services state and resources usage. If validation fails, additional services may be started to redistribute the load, resources may be scaled up and services can be mirrored or migrated to a different region in order to increase availability.

Cloud orchestration can be split into two approaches

- **Orchestration of software services** Services are orchestrated as executable business processes that interact both with internal and external SaaS services.

- **Orchestration of hardware services** It consists in computing resources management to satisfy cloud provider's operation requirements. Service Level Agreements are specified and abided by dynamical workload and resource reallocation.

In order to combine a central cloud with an edge architecture, many services will have to be coordinated, which can be simplified by using an orchestrator. After the service level agreements are defined, the orchestrator will find the most advantageous service distribution so that the system complies to the agreements. There are several orchestration platforms available in the market, being some of the most prevalent ones introduced bellow.

2.3.1 Docker Swarm

Docker Swarm [2] , is an open source orchestration layer integrated with Docker. It provides cluster management with a small learning curve to someone well acquainted with Docker. A cluster is called a swarm and is composed by nodes, being each node an instance of the docker engine running in swarm mode. There can be one or more nodes running in a physical computer, although not common in production environments. A node can be of two types: a manager node or a worker node.

Manager nodes perform orchestration and management functions required to maintain the desired state of the swarm. A single leader is elected by manager nodes (using raft consensus algorithm) to conduct orchestration tasks. In addition, a manager can also function as a worker node, but it can be configured to perform only manager related tasks.

Worker nodes receive and execute tasks dispatched by manager nodes. It reports about the task it has been assigned to manager nodes so that the manager can maintain the desired state of each worker.

[2]Docker Swarm Website - https://docs.docker.com/engine/swarm/ - accessed 06 February 2020

A service is a definition of the tasks to be executed, as it specifies the container image and the commands to be executed inside running containers. It is submitted to the manager node, resulting in dispatched tasks to worker nodes.

Moreover, a task cannot be reallocated to another worker and it can only have one of two outcomes, succeed or fail.

A service can be from one of two types:

- **Replicated Service** A replicated service replicates a task a certain number of times, the same as specified by the replication degree of the task.
- **Global Service** A replica of the task is run on each available node from the swarm.

2.3.2 Kubernetes

Kubernetes [3] , is an open source orchestration platform released by Google in 2014. It automates the scheduling of application containers within and across available computer clusters, never tying them to a specific machine. It is currently the most popular choice for container orchestration [16].

All its operation revolves around trying to achieve a *desired state*, specified in a declarative way by the developer. A desired state, indicates what applications to run, the intended number of replicas, what container images should be used and the amount of resources that should be made available.

2.3.2.1 Cluster

A Kubernetes Cluster is composed by a master node and one or more worker nodes. The master is tasked with managing the cluster, scheduling applications, maintain applications' desired state, rolling out updates and scaling applications if needed.

The worker node is a virtualised or physical computer in which the cluster applications are run. Each node has got a kubelet, an agent responsible for managing its node and interacting with the master node. The nodes communicate with master node using the Kubernetes API, exposed by the latter. The same API can be also used by end users wanting to directly interact with the cluster. A container run-time is also present in every worker node, allowing for container images to be pulled from a registry, container setup and execution.

2.3.2.2 Pods

A pod is the smallest abstraction unit present in Kubernetes, it models a logical host wrapping one or more application containers (see Figure 2.4). Some resources are shared between a given pod's containers, such as, storage volumes, a unique IP address and information about how to run each container.

Pods run in nodes and can never be subdivided, that is, its containers can not be split and reallocated to another Pod.

[3] Kubernetes Website - https://kubernetes.io - accessed 25 January 2020

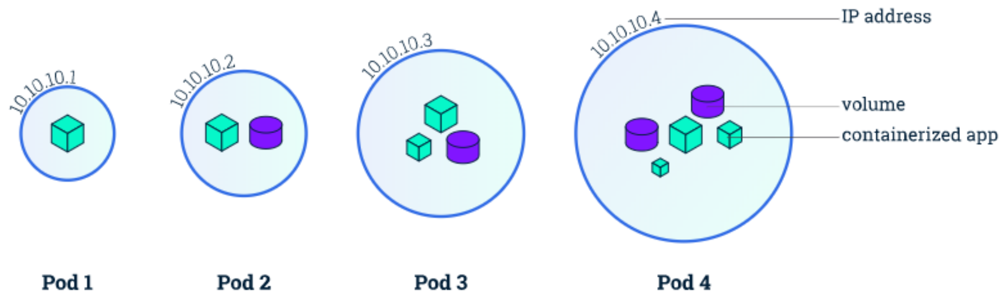

Figure 2.4: Kubernetes - Pod [17]

2.3.2.3 Deployments

Deployments are stateless high-level objects when compared to Pods. Kubernetes tries to maintain the desired number of replicas. The presence of replicas facilitates container image updates, while some of Deployment pods are being updated, the rest stay running, thus enabling zero down time. If the replicas are running in many nodes, the application can even withstand one of those nodes unavailability.

2.3.2.4 StatefulSets

Contrary to deployments, which are stateless, therefore interchangeable, where consecutive requests from the same client, can end-up in different pods, StatefulSets are needed in workflows where state preservation matters. For instance, for database management systems, video transcoding services, or truly any server which depends on state to perform its task.

In a statefulset, each pod receives a unique network name, which means traffic can be directly routed to a given pod, without depending on the volatile pod IP. If some pod happens to be recreated, it will be given the same name of the defunct.

2.3.2.5 DaemonSets

DaemonSet pods run on every cluster node. They're added and removed as nodes connect and disconnect from the cluster. They are often used for:

- Logging purposes
- Container Networking pods
- Storage mounting

2.3.2.6 Services

Pods are ephemeral, they are started, migrated and killed. Hence, keeping track of their network addresses presents some challenges.

An abstraction is needed to decouple pods that require the services of another set of pods (a sort of backend), without the need for the first to run its pods discovery service. A Service is introduced as an abstraction which groups a logical set of Pods and a policy by with they can be accessed. It also performs load-balancing on replicated pods.

There are four types of services, each of them is a superset of the previous:

- ClusterIP - used in intra-cluster communications, assigning the service a cluster internal IP.
- NodePort - exposes the service both internally (cluterIP) and externally to the cluster. The service is accessible as *<NodeIP>:<NodePort>*. A free NodePort is assigned automatically by the orchestrator but a particular one can be also specified, although this can lead to port collisions.
- LoadBalancer - support for external load balancers in cloud providers.
- ExternalName - exposes a service using an arbitrary name (specified by the services configuration attribute, externalName) wich returns a DNS CNAME record. No proxy is used in this type of service.

2.3.2.7 Persistent Volumes

Similarly to simple docker containers, pods have volumes — the location where data is stored which isn't part of the container image – that are bound to a specific pod's lifetime if the pod is deleted or migrated, its volume is lost. Difficulties arise in scenarios when data originated from pods execution must be preserved.

As a solution to this issue, there are PersistentVolumes (PV). They supplant pods lifetime, keeping its data safe. PVs can be statically provisioned (specified by the cluster admin), or dynamically created. There are many Volume Plugins enabling the provisioning of volumes compatible with different cloud providers and hosts. Pods interact with PVs mounting them, by the means of a PersistentVolumeClaim. These can be mounted by several pods allowing for data exchange (depends on volume plugin compatibility).

2.3.2.8 Pod discoverability - CoreDNS

CoreDNS is another CNCF project, which provides name server functionalities to a Kubernetes Cluster. It enables pod discoverability through its service's name, which is translated by a CoreDNS pod for the service's IP. The service then selects one of its backends, returning their podIP, thus concluding the discovery process.

2.3.3 Comparison

Docker Swarm is an easy to use orchestrator, well integrated with the Docker ecosystem. Whith its simplicity comes a rather reduced feature set.

On the other hand Kuberbetes brings with it very compelling features. This combined with its wide adoption and finer grained control makes it a great choice.

2.4 Monitoring

Monitoring in a cloud environment is really important, as it enables to evaluate and improve services. It allows to measure several metrics based on resource usage, keeping a record of said metrics, for future analysis. Those logs contain information used to optimise and calibrate the system's performance.

Some import concepts in monitoring are quality of service and quality of experience. Quality of Service is a concept usually related to network performance based of common network metrics: throughput, latency, bandwidth. On the other hand Quality of Experience is more complex, as it takes into account more than a mere technical perspective, by considering the end-users' subjective perceived quality [28].

oMon is a monitoring framework developed by Li et al. [18]. It implements monitoring functions on the VMM layer, thus being completely operating system agnostic, unlike many other solutions which require the installation of monitoring agents inside guest OSs.

It has got five components: the process monitor, syscall monitor, disk I/O monitor, network monitor and the correlation and analysis component (which uses data from the previously mentioned components to generate a comprehensive analysis).

Taherizadeh et al. [26] present a new approach in the field of Edge Cloud Monitoring, tailored to real-time applications, such as, online gaming, telemedicine services, environment monitoring systems and video conferencing.

The authors chose to monitor four QoS metrics, throughput, delay, jitter and packet loss, considered as important parameters in various video/audio streaming applications. Each Edge node runs a monitoring service that collects the QoS parameters between itself and each client, afterwards the data is sent to monitoring server. The monitoring server then reasons about the collected data, taking action if a given user's QoE could be improved upon the change of the node it is connected to. Depending on the use case, the QoS parameters weight can be modified by altering each of the associated scalars value.

2.5 Container Networks Interfaces

Container Network Interface is a generic plugin based specification for application containers backed up by CNCF. Different CNI plugins present different solutions to the networking problem and interface configuration. It is commonly used with orchestrators, to provide or enhance networking capabilities. Flannel [4] , Calico [5] , Kube-router [6] and WeaveNet [7] are four CNCF endorsed CNI plugins.

2.5.1 Flannel

Flannel is one straightforward solution which is simple to configure, capable, and suitable for most environments [12]. It is developed by CoreOS and consists on a Layer 3 (OSI Layer Stack - Networking) overlay network where each node is given a sub-network to administer and allocate its pods within.

Flannel has two main modes of operation, UDP and VXLAN (the default)

Ellingwood [12] recommends its utilisation as a starting point, then changing to another CNI if features needed are lacking.

2.5.2 Calico

Formally named Project Calico, it takes a OSI layer 3 approach using BGP (Border Gateway Protocol) routing protocol, which can route packets between hosts through the physical layer, dismissing the need for extra encapsulation, present in an overlay based solution. It also implements network policies, enabling pod network authorisation, based on an additive rule system.

Moreover, there is an hybrid solution combining Flannel VXLAN with Calico's network policy, called Canal.

2.5.3 WeaveNet

Weave Net by Weavworks, creates a mesh overlay network [12].

It has UDP multicast support and includes a DNS Server. It also has Network Policies and encrypted traffic support.

2.5.4 Comparison

Fannel (5500 stars, 1700 forks) is the most popular, based on Github repository stars and forks, followed by WeaveNet (5900 stars, 582 forks), Calico (1900 stars, 496 forks) and Canal (634 stars, 94 forks) as of 2nd of July 2020.

Kapočius N. [16] tested the performance of CNI options: Flannel, Calico (VXLAN, IPIP, and pure IP), Kube-router and Weave. The author came to conclusion that overlay techniques add significant delay, while pure IP approaches (Calico IP, Kube-router) have less introduced delay when compared to barametal tests. Flannel is the less delay inducing overlay plugin.

Duscatel A. has comprehensive CNI benchmarks with Kubernetes published online [11].

[4]Flannel GitHub repsitory - https://github.com/coreos/flannel/ - accessed 2 July 2020
[5]Project Calico Website- https://www.projectcalico.org/ - accessed 2 July 2020
[6]Kube-router Website - https://www.kube-router.io/ - accessed 2 July 2020
[7]WeaveNet Website - https://www.weave.works/oss/net/ - accessed 2 July 2020

Figure 2.5 shows Cilium correctly detecting the hosts configured MTU, and Flannel is just off by 1%. Having the MTU auto detected eases the CNI's instalation process.

Figures 2.7, 2.8 and 2.6 compare varied CNI's performance on TCP, UDP and HTTP, three of the most used protocols on the Web.

Duscatel concludes, that for low resource nodes, without advanced security features (encryption and network policy) — our validation cluster case — Flannel is the right choice as it is performant and of easy configuration. Otherwise if improved security is a must, WeaveNet is the one, however introducing a hefty overhead.

Flannel was chosen as the CNI for this project.

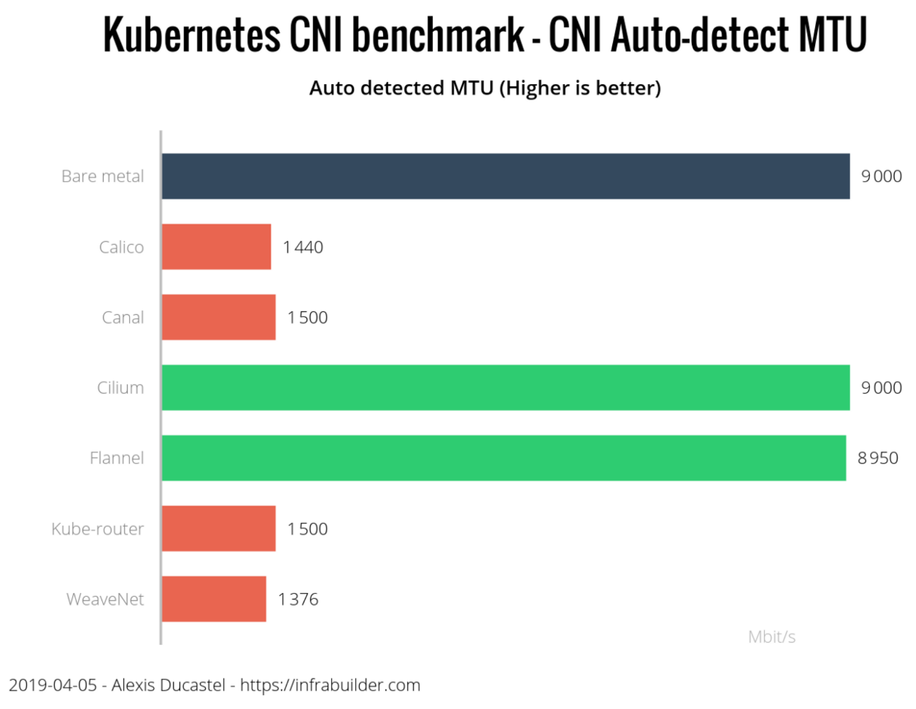

Figure 2.5: MTU auto-detected by CNIs [11]

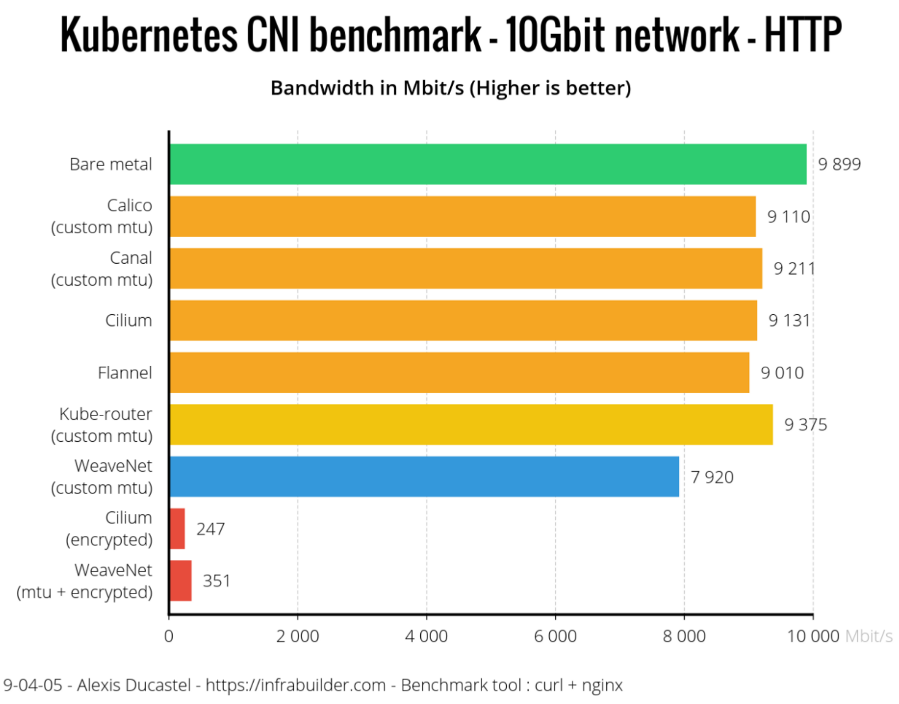

Figure 2.6: CNI HTTP bandwidth [11]

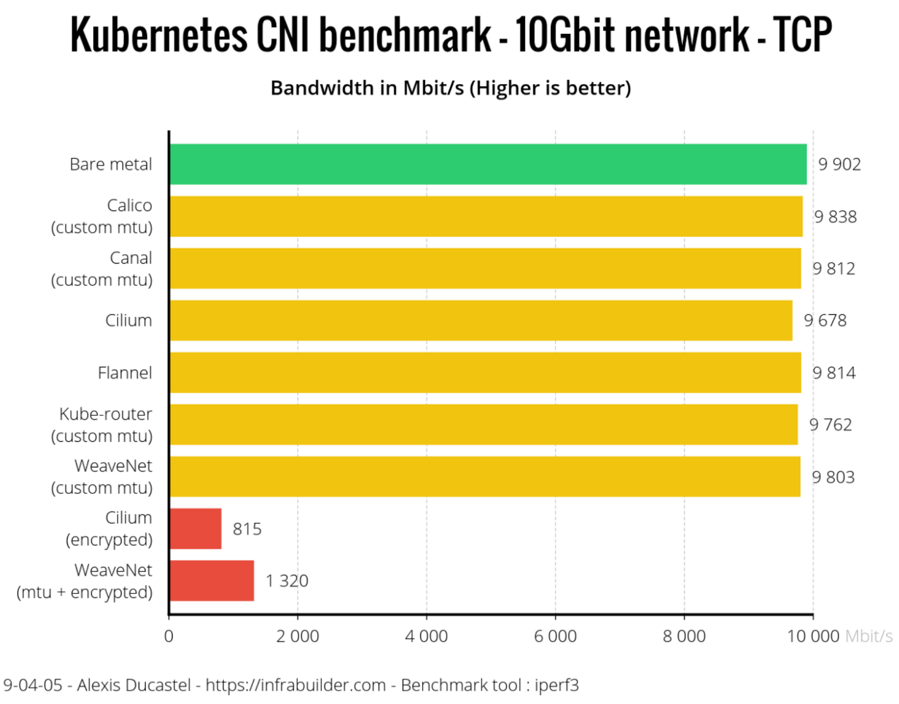

Figure 2.7: CNI TCP bandwidth [11]

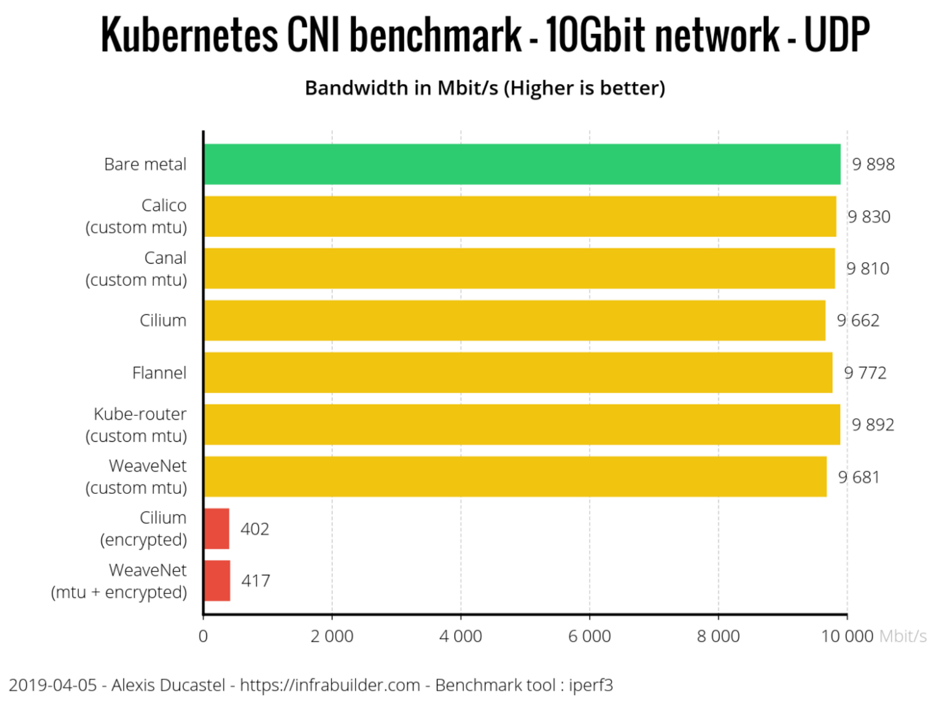

Figure 2.8: CNI UDP bandwidth [11]

2.6 Micro Service Architecture

Micro Service architecture is a modular, distributed and agile emerging architecture where services are small, isolated and cooperate to achieve a complete system [16]. It contrasts with a monolithic architecture where the system is tightly coupled, and harder to scale [2]. As each service can exist by itself, this means separate development teams can work on their service individually, without constant input from the other teams. They only have to agree on a common communication API.

Cloud systems many times follow a micro service architecture as this lets them make better use of available resources, by scheduling containers to hosts which better suits their resource requirements [15].

However, in large complex projects the absence of a clear, global architectural plan can lead to services identities and dependencies to become less precise [2].

In [13], M. Fowler et al. analyse the differences in scaling a monolithic application when compared a micro-services architecture. Figure 2.9 illustrates such differences. The authors also state that the tight coupling in a monolithic approach, hinders flexibility to perform updates in a cloud environment, as every required change requires the whole application to be rebuilt and re-deployed.

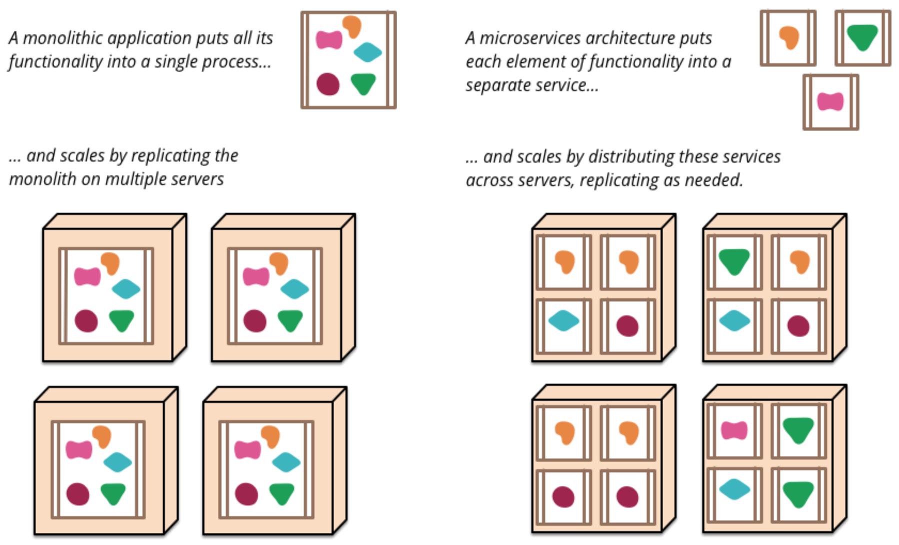
A monolithic application puts all its functionality into a single process...
A microservices architecture puts each element of functionality into a separate service...
... and scales by replicating the monolith on multiple servers
... and scales by distributing these services across servers, replicating as needed.

Figure 2.9: Scaling Monolitc and Microservice Applications [13]

2.7 Crowdsourcing

Crowdsourcing can be defined as any kind of task where non professionals, contribute sharing their knowledge and expertise for a common goal [1]. The crowd contributes voluntarily or in exchange for compensation. It gathers the collective knowledge on a topic, hence it isn't bound to a specific field.

Coursourcing can be applied to Journalism, where the users can give their insight on a given theme or topic, then a journalist could use the gathered crowd information, to produce a news piece that has a more impartial, transparent, and inclusive take on the subject. Otherwise perhaps it would simply not exist due to lack of sufficient information. However in crowdsourcing, information or work performed won't be as coherent as if it had been performed by an individual or a team with an aligned set of processes and norms [1].

This concept can also be applied to media streaming technologies where crowd generated content can be used to compose multiple views of a concert, for instance, where stream's spectators could view adjacent views, this way enjoying a more immersive experience [4].

2.8 Media & Networking

Media recording and playback are quintessential for the development of a multimedia system. Along with it, networking technologies are key in the efficient transmission of audio and video between end-users and the cloud content provider. Therefore, the study of these fields is highly important to achieve the proposed objectives, a performant solution possesses quality coupled with low resource consumption.

WebRTC and RTP are analysed as they are highly relevant in the real-time live streaming panorama.

2.8.1 WebRTC

WebRTC is an open standard for video, audio and generic data (eg. text, files) real-time communication. It is supported by modern browsers through a JavaScript API and native applications conforming to the specification [6]. Communications are peer to peer over RTP, mediated by a *PeerConnection*, without the need of an intermediate server. A PeerConnection is established by a signalling protocol (eg. Session Initiation Protocol [SIP]).

It requires both peers to be on the same local network (ie. reachable without NAT), otherwise a Session Traversal Utilities for NAT (STUN) server can be used to help with the signalling protocol and a Traversal Using Relay NAT (TURN) to bypass firewall and NAT complications.

2.8.2 Realtime Protocol (RTP)

RTP is a transport protocol for real-time applications, audio video or data over a unicast or multicast connection.

RTP packet has base fixed header, containing most notably: a sequence number, a timestamp and a payload type header. Moreover, the protocol is extensible allowing for individual implementations to add extra functionalities, said functionalities require extra information to be supplied as an extra header, the extension header.

Monitoring and control flow like functions are addressed by Realtime Control Protocol (RTCP) [23]

2.9 Summary

This State of the Art examined many fields relevant to the development of an edge cloud architecture. The NIST's cloud computing model was presented defining its main characteristics and concepts. Afterwards, Edge computing was analysed, comparing it to a regular central cloud, asserting it as better suited for applications that have got strict networking restrictions (low latency and high bandwidth), ones that require a high degree of interaction and where users produce a large volume of data.

Virtualisation was defined due to its importance, by allowing for failure isolation, resource isolation, cloud scalability as well as resource monitoring. Subsequently, virtual machines, containers and unikernels were discussed and compared, emerging the containers as the best compromise between efficiency and ease of use.

Orchestration addresses the problem of coordinating software and hardware in a cluster, replicating services to provide high availability. After a desired state is specified, the orchestrator performs actions in order to close in the gap between the current state and the desired one. Docker swarm delivers a easier to use experience to a person already acquainted with Docker, while Kubernetes has wide adoption in large and complex projects due to its richer feature set and finer-grained configurability.

Monitoring involves component evaluation to keep a record of their events, resource usage and other metrics. These metrics are used to optimise the system's performance and to know what happens within it. Taherizadeh et al. present a novel approach for real-time applications on Edge Cloud, it uses QoS parameters to determine the best edge node a user can connect to. Li et al. developed a framework, which operates on the VMM layer, capable of monitoring without the need for monitoring agents in the guest OS.

Container Network Interfaces is a generic networking specification tailored to containers. Each of the different implementations has its own pros and cons, making their efficacy dependent on the use case.

Crowdsourcing can be summarised as amateur contributions to a specific end, be it a news article of funding a product or idea. The use or the collective knowledge contributes to a more global an inclusive perspective on a give subject.

Micro Service architecture brings responsibility separation at the architectural level. It brings better scale prospects to a cloud application, as selected modules can be scaled up, instead of having to forcefully scale the whole system, this makes better use of available resources.

WebRTC and RTP were also analysed due to their relevance for livestream video infrastructure implementation.

Chapter 3

Proposed Solution

Based on the analysis conducted of the relevant state of the art and on the associated comparisons presented in the previous chapter, it was possible to take decisions concerning the approaches to adopt and the technologies to select to conceive a solution that would satisfy the identified goals. More specifically, the following decisions were taken:

– Container based virtualisation was chosen as the virtualisation approach by the virtue of its ample utilisation, quick startup (a highly sought after attribute in a cloud computing environment), and good synergy with a Micro Service architecture.

– Kubernetes emerged as the right choice, concerning orchestration, due to its rich ensemble of features, high popularity, hence leading to the existence of good documentation.

– The chosen CNI was Flannel in VXLAN mode, as it provides a great compromise between ease of use and performance, falling only shortly behind solutions which require extra tuning to achieve their full potential.

– WebRTC is utilised for the ingest part of the multimedia data path, on account of its reduced latency and widespread adoption in mobile devices. It should be noted that a company related requirement lead to RTMP use on the final portion of the same data path.

This projects main use case is a large event live coverage. For instance, in a Music festival scenario, a news agency could benefit from several views covering the event, recorded from different perspectives akin to multi-view video. The edge segment would be launched at the start of the event, live video would be processed at the edge servers, making use of the very low latency and high bandwidth available at the venue's local area network. The edge component would scale automatically as seen fit by the application. Additionally, the live videos would also be recorded to disk and kept at the edge during the event's duration

At the end of the event, the edge segment can be teared down as it is no longer needed, for the time being. At the same time, the event recorded footage can be uploaded to a central data-centre for safe storage, without any realtime requirements.

This chapter starts by providing some background information on previously existing developments that have implications or impose restrictions on the proposed solution and that, together with it, form the overall system capable of implementing crowdsourcing live journalism services. It then provides a high level description of the complete system, proceeding to detail the different components of such proposed solution.

3.1 Background

Mogplay is a video streaming platform developed by MOG Technologies. It is a platform suitable for many use cases from enterprise video market, large event coverage, to a user-centric video social network (similar to Periscope [1]).

This work focus on the Crowdsource Journalism configuration, where regular non-professional users, transmit their live video/audio feeds to the news company (where the solution is located) in exchange for a monetary reward. As in everything where money is involved, there has to be trust in the system's security and reliability. Therefore, undertaken transactions have to be accessible in a transparent manner while remaining immutable (*ie.* once performed, they are appended to the log and become irreversible). These proprieties are guaranteed by the use of a blockchain, where a new block is concatenated to the previous chain of blocks, each one representing a transaction approved by the interested parties through a consensus algorithm.

Mogplay records mobile users streams in persistent storage at the same time that transcodes the live feed, standardising it, so it can be used in live broadcasting. All the streams converge on the multi-viewer, where a broadcasting operator chooses which one will be the system's output, at any given moment. The output stream can then be transmitted to a traditional broadcast room for further editing or directly relayed to end viewers.

3.2 Stream Flow

The streamer visits controller website and can login as a journalist, a regular citizen or as an anonymous user, without the need to authenticate. After choosing one of the above the user is redirected to the stream page, where it possible to initiate a stream, by pressing the red record button. At that moment the stream initiation process is triggered, depicted in Figure 3.1. The streamer's browser sends a *startRecording* request to Media Controller's backend, which in turn requests Resource Manger for a free transcoder IP. Concurrently a WebRTC session is establish between the mobile and WebRTC server. Afterwards, when the new transcoder is ready, its IP is returned the controller and relayed to the streamer's browser. Finally, the allocated transcoder IP is sent to WebRTC server, so the server can relay the incoming audio/video stream to the appropriate transcoder.

[1]Periscope - A livestreaming platform for iOS and Android, owned by Twitter - `https://www.pscp.tv/` - accessed 18 June 2020

Succeeding the streams transconding, they're sent to the video switcher via RTMP, where they become available for consumption.

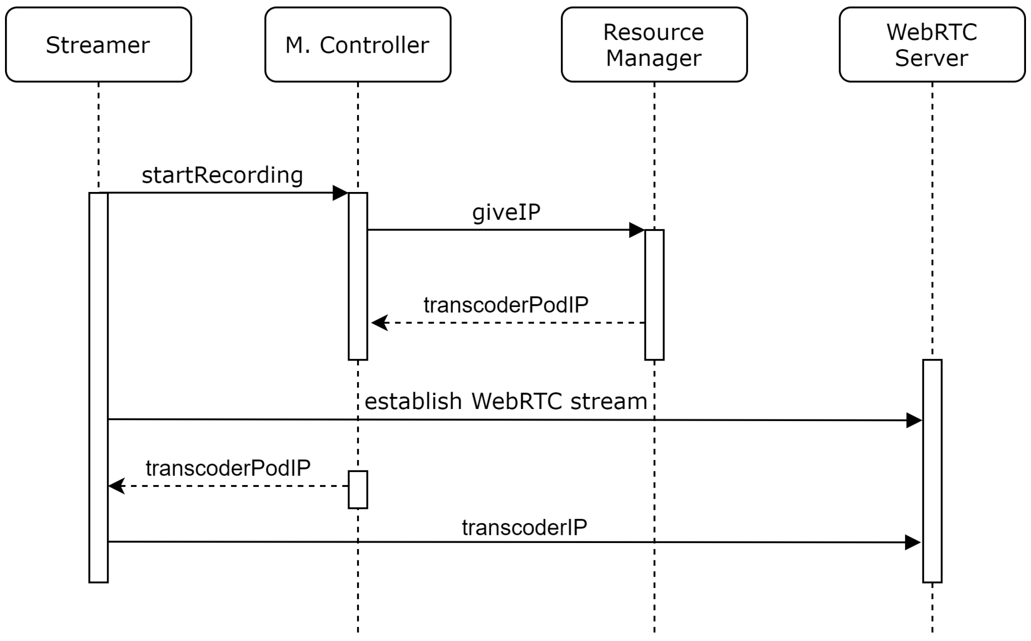

Figure 3.1: Stream Flow

3.3 Architecture

Mogplay is based on a micro service architecture, so that every component is self contained, enabling its exchange should the need arise. This kind of architecture also allows for a specific component's replication, without the need to scale up the whole system. The system is composed by three subsystems, the streaming subsystem, recordings and the broadcast subsystem. Each of the three subsystems is in turn subdivided into its basic services.

The basic services are translated into Kubernetes' Pods or equivalent higher level constructs, inter-connected by the CNI in a multi-node cluster.

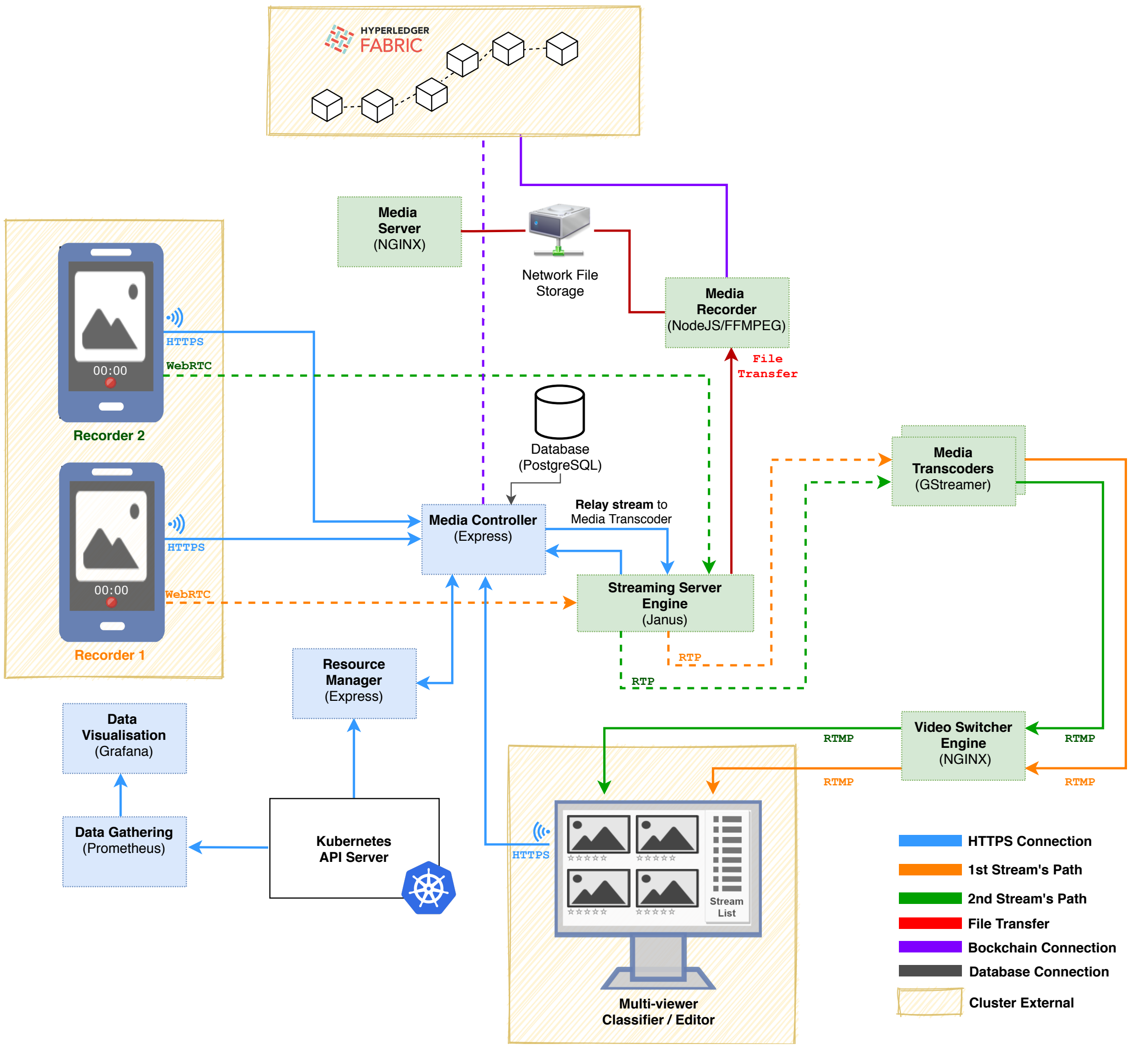

Figure 3.2: Mogplay Architecture

Figure 3.2 depicts the services pertaining to Mogplay. Bellow each of them has its purpose and responsibilities explained. From the figure it's possible to understand how a stream unfolds following the data path form the Recorders to to the Multi-Viewer.

1. **Media Controller**

   This component is responsible for managing main application level operations, such as: creating/stopping streams and user authentication. It is where the streamer and the multi-viewer operator web GUIs reside.

   It receives and transmits requests trough an Express [2] HTTP Web server and Web Sockets. The HTTP server is accessible externally through the cluster ingress while the Web Sockets are solely for intra-cluster message exchange.

2. **Resource Manager**

[2]Express - A minimalist web framework for NodeJS - https://expressjs.com - accessed 27 June 2020

Formed by an Express Web server serving a HTTP API, it hands out transcoders and keeps track of their availability. Additionally, if there is no free transcoder to be allocated to a new stream, resource manager issues a Pod creation asynchronous request to Kubernetes API Server. It waits for the transcoder full initialisation before relaying the new pod's podIP to the request's issuer, Media-Controller.

Table 3.1: Resource Manager's HTTP API

Route	HTTP Verb	Description
/giveOne	GET	Gives a free transcoder, creating one if required
/freeOne	GET	Free the specified transcoder, scaling down the transcoder count if adequate
/listVMs	GET	Lists transcoder list
/pods/all	GET	Lists all pods

In Table 3.1 it's possible to see resource manager's HTTP API, detailing the route, HTTP method to be used and a short description of its functionality.
Going into a little more detail, */listvms* response has a JSON structure akin to Listing 3.1, where for every transcoder pod its pod IP, node IP (hostIP), usage flag, pod name and streamer hash are listed.

```
{
    "10.244.1.120": {
        "podIP": "10.244.1.120",
        "hostIP": "10.00.40.208",
        "inUse": true,
        "name": "transcoder-3-axjc9h",
        "hashUser": "hpaU2botIDGiMUVhAAAA"
    },
    "10.244.1.122": {
        "podIP": "10.244.1.122",
        "hostIP": "10.00.40.208",
        "inUse": false,
        "name": "transcoder-2-swho9e",
        "hashUser": ""
    },

   ...
}
```

Listing 3.1: Endpoint */listVMs* response example

At startup, Resource Manager issues the creation of a preset number of transcoders, currently set to 3. For ease of use instead of performing raw HTTP requests to interact with Kubernetes API, a request wrapping NodeJS library was used, @kubernetes/client-node which in turn makes the following request POST */api/v1/namespaces/{namespace}/pods*.

Moreover, this module has subscribed to any pod deletions, mainly to handle issues related to transcoder pod crashes, so if anything happens Kubernetes will inform it. Afterwards, a new transcoder will be created to take the old one's place, this will be further explained in 3.4.

3. **Streaming Server - Janus**

   Implemented using Janus [3], "a general purpose WebRTC server" as stated by its creators, whose modular design — extensible using Meetecho's provided plugins or even community developed ones — meant we could tailor our server to our needs, while keeping its footprint rather small, only including functionalities we really needed.

   It's the receiver of the several mobile streams, functioning as a WebRTC Peer. After initiating a new WebRTC stream successfully, the server is instructed to relay received RTP streams (audio + video) to the controller designated transcoder. Where the audio and video will be processed to better suit the event coverage purposes. Such purposes would be that every livestream would have the same encoding, resolution and framerate so when the broadcast stream was switched the experience would remain smooth.

4. **Media Transcoder**

   Each transcoder service, receives two RTP streams (1 audio, 1 video) from WebRTC Server, which are then processed by a GStreamer [4] instance, resulting on two normalised streams destined to the Video Switcher which afterwards can be broadcasted through the Multi-viewer if its operator so desires. The requests to start transcoding are handled by a simple Express Web Server.

5. **Video Switcher**

   Receives every RTMP stream after having been processed by transcoders, in turn makes them available to any compatible player. Media Controller is who controls which stream is broadcast to end viewers, using HTTP requests through the Multi-Viewer. It also has */stats* with some debug information about currently active streams.

   It's accessible by a NodePort service, enabling both internal and external communications with the cluster. HTTP external connections are delivered to the service by the Ingress Controller, which redirects packets destined to `https://switcher.mogplay.local`

[3] Janus - `https://janus.conf.meetecho.com` - accessed in 24 June 2020
[4] GStreamer - An opensource media framework - `https://gstreamer.freedesktop.org/` - accessed in 24 June 2020

6. **Media Recorder**

   This component handles the stream recording process, by saving to "disk" (a Persistent Volume 2.3.2.7) every WebRTC's RTP packet is received as a partial file. These files can then be merged into a single video container file (eg. mp4, mkv). To benefit from our Edge approach media recorder files are kept at the event site, until event termination, when partials files can be combined and the resulting files transferred to the central cloud for definitive storage. It should be noted that after the event, our system real-time needs can be relaxed, due to absence of streams of any kind.

7. **Media Server**

   Media server, implements Access Control to the recordings NFS share. A video can only be downloaded upon supply of a authorisation token, which directly identifies the intended recording, giving access to it for a limited time.

   For this purpose, a NGINX [5] instance was used.

8. **Multi-viewer**

   It is operated by a broadcasting producer authenticated in the platform. The producer can preview the current streams, dragging them from the list to four larger viewers, and choose to which of the them to be the final broadcast.

9. **Application Database**

   It stores application related data, such as: users, events, platform permissions, in a PostegreSQL [6] relational database management system.

10. **Blockchain ledger**

    The blockchain is where both stream related metadata and video purchases are stored. There are only two components which write to the blockchain, the Media Controller at the start of a stream — information streamer's identity, geolocation, associated event — and the Media recorder when the stream ends, adds a stream termination timestamp.

11. **Streamer**

    The streamer is a regular user which uses their mobile device to record a livestream about some phenomenon in their close vicinity. Comunication with Media Controller happens over a secure channel, using HTTPS. Along with it the mobile establishes a WebRTC connection with the WebRTC server, for audio and video transmission.

12. **Prometheus**

[5] NGINX - `https://www.nginx.com/` - accessed 26 June 2020
[6] PostegreSQL Website - `https://www.postgresql.org/` - acessed 5 July 2020

Prometheus [7] is a time series data base (TSDB) [8] created by SoundCloud in 2012, and part of the CNCF since 2016. It deals with most steps of the monitoring flow, apart from data visualisation, such as: metrics gathering (through endpoint scrapping), data storing, alerting, and data querying (using its own query language, *PromQL*).

Our prometheus scrapes metrics from two services:

- NodeExporter (a DeamonSet) exposes OS and host hardware metrics such as: cpu, memory, processes, disk I/O, filesystem usage.
- Kube-State-Metrics (a Deployment) exposes cluster metrics, such as: node information, pods resource utilisation and other Kubernetes object information.

13. **Grafana**

    Grafana [9] addresses data visualisation, enabling the creation of user-friendly real-time dashboards, which help us get a better grasp of resource bottlenecks and system anomalies. It is compatible with different data sources, from relational (SQL), non relational to timeseries databases. Additionally, it allows queries to be embedded into the dashboards, this way, speeding its creation process.

3.4 Fault Tolerance

Kubernetes pairs up well with a micro-service approach to system architecture. Such systems can become fault tolerant, that is, to carry on working in case of one of more services failure, recovering automatically. Using higher level abstraction, instead of plain pods, ensures the orchestrator can have more insight on your pod requirements, due to their more semantic attributes. If an pod fails, Kubernetes won't recreate it and reschedule it automatically, unless this pod was managed by a Deployment or a StatefulSet. In Mogplay every component is a Deployment or a Statfulset, except from Media Transcoders — as they are created and deleted by Resource Manager — and the blockchain (which is external to the cluster). That means that if any of those crashes, the orchestrator will recreate them, thus repairing the missing "coggs in the machine".

Resource manager, in order to prevent stream failure upon transcoder crash, assigns a new transcoder pod therefore keeping the stream live. It subscribes to transcoder pods deletion events, making sure that the transcoder is replaced if it was active, also notifying media controller. Otherwise it just issues a new transcoder creation request as is depicted by Figure 3.3. Media Controller's main server upon receiving a *transcoderFailed* warning from resource-manager, proceeds to propagate it to the correct streamer. Finally, this streamer asks Janus to update the first's RTP redirection destination to the new transcoder IP. As a results this automatically recovers the disturbed stream. The same functionality could be extend to accommodate the change of a stream's

[7] Prometheus Website - `https://prometheus.io` - acessed 23 June 2020

[8] Time Series Data Base - A database management system that records discrete values thought time. This data is suitable for graph plotting

[9] Grafana Website - `https://grafana.com` - accessed 23 June 2020

transcoder for other reasons, such as to upgrade to a more capable transcoder better suited for higher quality video.

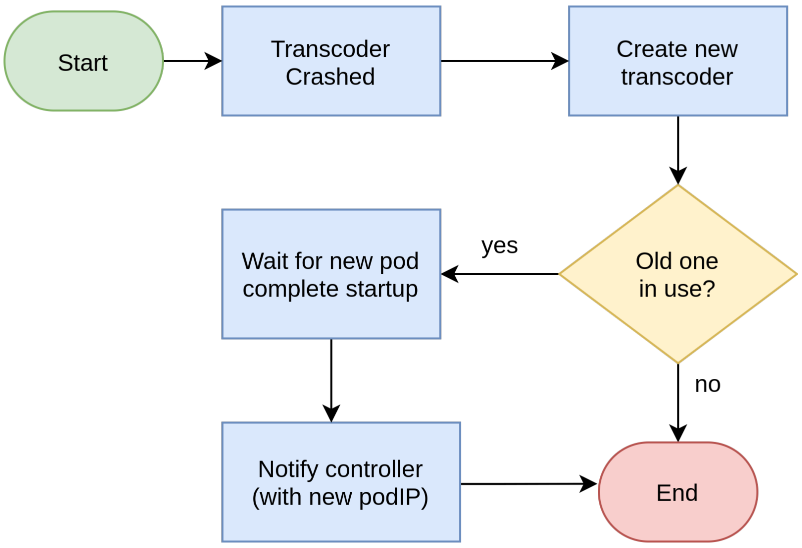
Start
Transcoder Crashed
Create new transcoder
Old one in use?
yes
Wait for new pod complete startup
no
Notify controller (with new podIP)
End

Figure 3.3: Resource Manager - Transcoder Recovery

3.5 Monitoring

As stated previously, Prometheus is used to collect and save cluster and host nodes information, which can be used to get a better insight the systems resource usage.

A Grafana dashboard was created to consume Prometheus data, this way making it more easy to be comprehended. Figure 3.4 shows some of the existing graphics. Such graphics were used to take bandwidth, CPU and memory consumption data, described in Chapter 4.

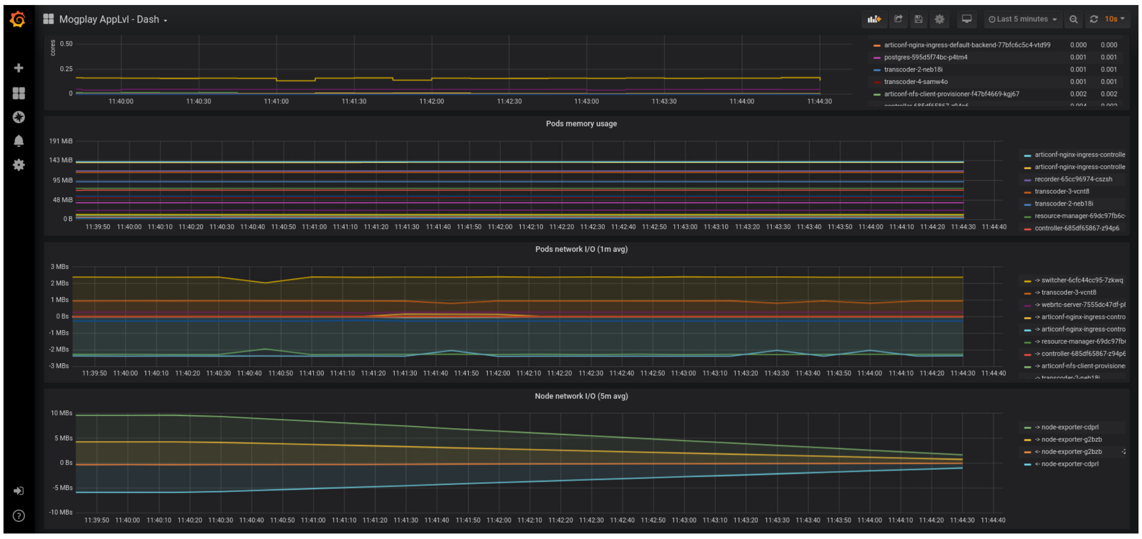

Figure 3.4: Developed Grafana Dashboard

3.6 Cluster Networking

Pods communicate differently, depending whether the requests are internal, between cluster services or if they're external to the cluster,

3.6.1 Internal

Pods internally communicate using services and podIPs from all nodes without using NAT. Inter-node communications are assured via an overlay network, provided by Flannel, following the CNI specification.

3.6.2 External - Ingress

An Ingress lets external HTTP/HTTPS traffic access the cluster in a controlled way. Traffic arriving at the ingress, depending on the **Host** HTTP request header are then routed to its corresponding kubernetes service. NGINX Ingress controller is the one being used.

Table 3.2: MOGPlay Ingress Routes

Route	Service
controller.mogplay.local	Media Controller
resource-manager.mogplay.local	Resource Manager
webrtc.mogplay.local	WebRTC Server
switcher.mogplay.local	Switcher
media-server.mogplay.local	Media Server
prometheus.mogplay.local	Prometheus
grafana.mogplay.local	Grafana

3.7 Storage

Most of the pods information is volatile as there is no need to keep information about most processes, as they are used to transmit, process or visualise the live streams. These streams generate artifacts which have to endure for long after the system's shutdown, so they need to be kept on a PersistentVolume.

3.7.1 Network File System

A Network File System content endures pod termination and is writable by multiple pods simultaneously, which grants NFS great versatility. This data persistence means data can be preloaded to a NFS volume to later be access by pods mounting that volume.

Grafana data, prometheus collected data, and original (unprocessed) video recordings are kept on a network file system.

The NFS is accessible from any of the nodes, as long as they possess adequate utilities to mount an NFS share.

3.8 Development Environment

Kubernetes in Docker (Kind) [10] was used to facilitate the development stage. It supports multi-node cluster, similarly to what is commonly used in production environments. The nodes are virtualised using docker containers running on the host machine. Pods are then run against said nodes, which allowed to test some of the multi-node cluster features in a single computer. Additionally, it made easier the transition to a production level cluster on physical machines.

Helm [11] is a Kubernetes package manager, which it calls charts. Charts can be downloaded from remote repositories, allowing its publication and sharing. Along with it, Helm also groups deployment's yaml files giving it a name and version number, which simplifies its installation, deletion, and modification. Sub-charts can be added to a main chart as dependencies, akin to what happens in a regular packager manager. Furthermore it has got a templates engine, that processes chart templates at installation time, replacing template values by concrete values present in specified configuration files.

3.9 Limitations

Our solution currently only works if the mobiles and WebRTC Server are located on the same LAN. In order to make it possible over a Wide Area Network (eg. internet) which is most of the times behind NAT and firewall, a TURN and STUN server would need to be configured. We chose not to due to the possible introduced delay and inherently added complexity. Furthermore, the available bandwidth might not have been constant and sufficient for high quality video streaming

3.10 Summary

In this chapter implementation background was presented, the main use case described, the architecture was depicted, explaining each component's functions as well as a stream workflow.

Additionally, the solution's fault tolerance mechanisms were elucidated. namely the orchestrator's rescheduling of failing components, trying to reach the declared desired state, as well as the our custom fit stream recovery implementation.

Container Networking was described, cluster storage discussed and relevant development environment tools were listed.

The implementation limitations were also exposed and the reason of their existence explained.

[10] Kind - `https://kind.sigs.k8s.io/` - accessed 26 June 2020
[11] Helm - `https://helm.sh/` - accessed 26 June 2020

Chapter 4

Validation

This chapter describes the work that was conducted during this dissertation to allow assessing the validity of the proposed solution. To perform a more complete evaluation, it was decided to conceive two different testing environments, each one implementing/simulating a specific cluster configuration. The first one takes a more traditional cloud approach, deploying the solution at the company premises, that is, a centralised approach. While the second configuration splits the cluster into two sites, a central site (company premises) and an Edge site (at the event venue).

Firstly, it starts by describing the different setups, explaining their configuration, what sets them apart. Secondly, the procedures used to configure the cluster are described. Afterwards, measured metrics are shown, compared and evaluated. Finally, conclusions on the validation experiments are drawn and this chapter summarised.

4.1 Environments

In Environment 1 central cloud, every cluster node is far away from the event site, where the streamer and operator are, this will serve as the baseline for our comparison. This will be the case, as its the most common configuration on a cloud application.

Environment 2 has *Cristal* (master node) and *Imperial* (cloud worker node) at the company premises, while *Desperados* (edge worker node) is at the event location. We predict shortened delay and increased bandwidth between streamers, Janus and the transcoders, what will result in lower end to end stream delay.

Both of the environments are depicted in Figure 4.1 and Figure 4.2 where each edge displays the approximate existing delay between its extremities.

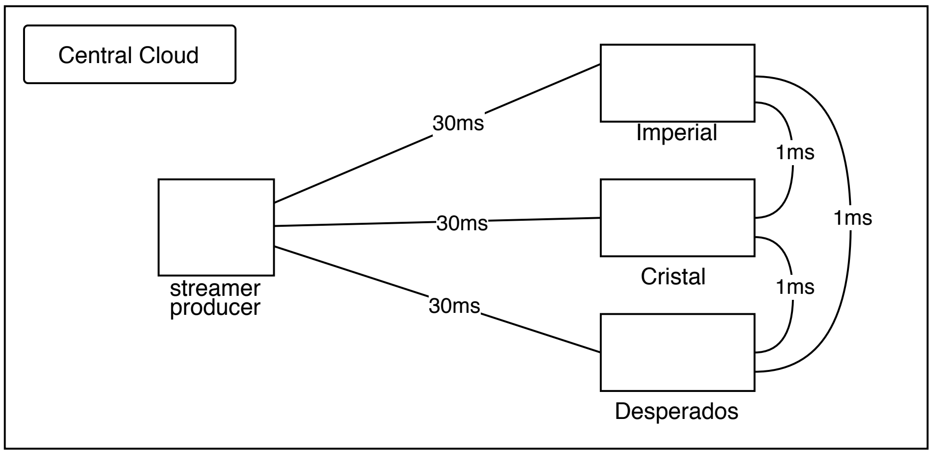

Figure 4.1: Central Cloud Environment

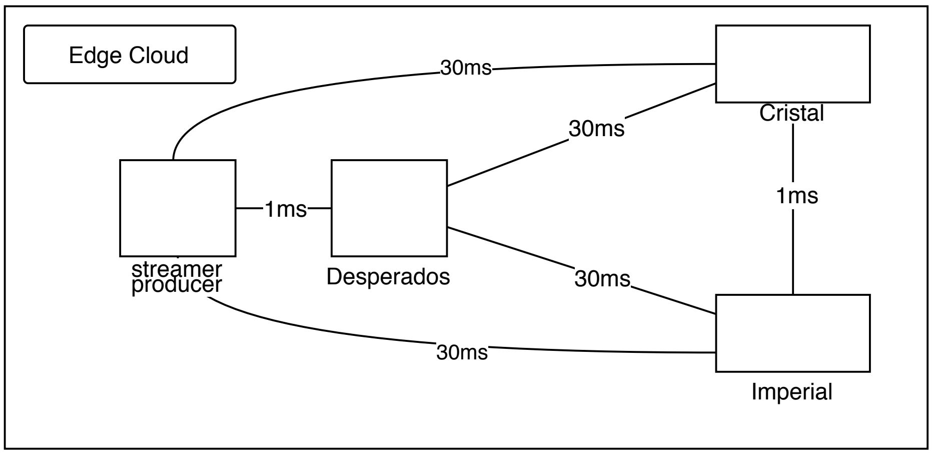

Figure 4.2: Edge Cloud Environment

4.1.1 Host Bandwidth Tests

All the computers are connected by a 1 Gbps switch. So to have and idea of the real bandwidth between the client and nodes, tests were performed using *iperf3*.

The results were slightly lower than the theoretical limit from Client: to Cristal (910 Mbps), to Desperados (907 Mbps) and to Imperial (917 Mbps).

4.1.2 Cluster Setup

Each of the three servers possesses a x86 Intel platform running Ubuntu 18.04 Long Time Support (Tables 4.2 and 4.3). Required Kubernetes dependencies and auxiliary CLI tools were installed (Table 4.1), namely Docker Container Engine, Kubelet, Kubeadm and Kubectl.

Docker is the previously discussed container engine in the State of the Art, subsubsection 2.2.2.1.

Kubelet is the program responsible for receiving and handling master control plane's requests, launching pods, and managing them, its execution is what makes a host a Kubernetes Node.

Furthermore, Kubeadmn (Kube Admin) bootstraps Kubelet's initial configuration. It aids with cluster creation, the creation of a master node, as well as with a new node's join to the cluster.

Finally, Kubectl (Kube Control) is a CLI that eases the cluster users interaction with Kubernetes API. It allows for Kubernetes object lists retrieval, object creation, edition and deletion.

The cluster was created in Cristal, using *kubeadm init*, followed by the execution of *kubeadm join* on the other two machines. It should be noted that Kubernetes does not provide an included CNI implementation, so subsequently, Flannel install configuration files were applied to Cristal, what resulted in a Flannel daemonset and consequnetly a Flannel CNI pod in every node. With this the VXLAN overlay network was set, bridging the nodes, thus completing the cluster setup.

Table 4.1: Validation Cluster Software Stack

Name	Version
Docker	19.03.6
Kubernetes	1.18.3
Kubeadm	1.18.3
Kubelet	1.18.3
Kubectl	1.17.3
Helm	3.1.0

Table 4.2: Cluster Machines Specifications

Name	CPU	Cores/Threads	Memory	Role
Cristal	Intel Core i7-2600	4/8	8 GB	Master
Desperados	Intel Core i7-4790S	4/8	16 GB	Edge
Imperial	Intel Core i7-2600K	4/8	8 GB	Central Cloud

Table 4.3: Cluster Machines Operating System

Name	OS	Linux Kernel Version
Cristal	Ubuntu 18.04.4 LTS	4.15.0-88-generic
Desperados	Ubuntu 18.04.3 LTS	5.3.0-59-generic
Imperial	Ubuntu 18.04.4 LTS	4.15.0-88-generic

4.1.3 Traffic Control (TC)

To validate our hypothesis, that an Edge cloud cluster has better performance than a full cloud configuration, geographical distance was emulated. This was done by introducing delay using *tc*, a Linux system administration utility used to show and manipulate kernel traffic settings. It was used to modify the CNI's network interface queue delay between the central nodes, edge node and the streamer & operator machine.

NetEm (Network Emulator) is a tc queuing discipline that allows to simulate various network delays or disruptions, some of them: packet corruption, delay, jitter, duplication, limited rate and reordering. Most of them can be configured based on a mathematical distribution or a percentage.

Tc filters were used to specify which were the destination IPs whose packets we wanted to delay, because not every packet being transmitted via an interface was intended to be delayed.

4.2 Results

Considering both scenarios presented above, the following metrics were gathered: Nodes bandwidth consumption, pods bandwidth consumption, end to end stream delay, Pod CPU usage and memory usage are discussed bellow and can have its source data consulted in Appendix A

A third simple configuration is used, deemed as optimal, nevertheless unviable for a real like scenario. In this scenario, every node and the client belong to the same Local Area Network. This brings with it a rather insignificant delay (lesser than 1ms) and high bandwidth, only impacted by the switch connecting the computers.

It worth noting that each transcoder consumes approximately 1 CPU thread, this means *Desperados* with its 8 threads,is only able to sustain at most 7/8 streams at a comfortable CPU utilisation level. Over this threshold, it will struggle to keep up with the demand.

End to end latency was measured from the moment the video is recorded at streamer's mobile, until it is watched on the multi-viewer, again outside the cluster. The values present in Figure 4.3 are the average delay in seconds having precision to the hundredths of a second. Furthermore, each scenario's bar is the average of n streams (at most four) being displayed on multi-viewer's four preview slots, whilst the rest still contribute to system stress.

Delays from one to four concurrent streams are very similar in every scenario, with Central Cloud having just bit more latency than the other two approaches.

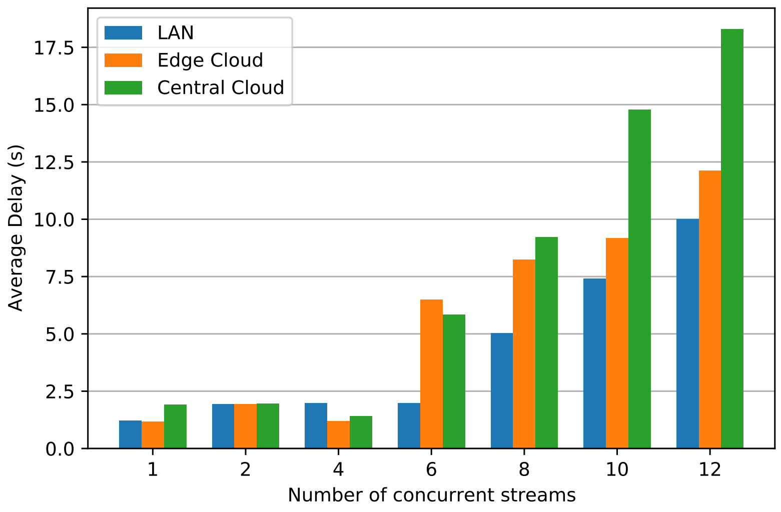

Figure 4.3: Average End to End delay

With the increasing number of streams, the the delay gap between the central cloud, the edge and LAN scenarios widens considerably. At twelve streams the central cloud has on average six seconds more delay.

Figure 4.4 shows the consumed pod bandwidth for the main media streaming pods: WebRTC Server, Video Switcher and one of the active Transcoders (as they all use the same resources, transcoding each one stream). Only these were plotted as the rest remained more or less constant bandwidth consumption, irrespective of the number of active streams.

With one stream, the Switcher is receiving 2.36 MB/s, increasing linearly until 6 streams are reached with values of approximately 14.2 MB/s.

Transcoder curves IN and OUT bear constant values, as they depict the values consumed for one only active transcoder, 0.58 MB/s and 2.34 MB/s respectively.

Additionally, the WebRTC curves represent the bandwidth used by receiving the mobile streamers and naturally the consequent RTP forwarded streams to each respective transcoder. It also increases linearly, but with a slower slope than the Switcher curves.

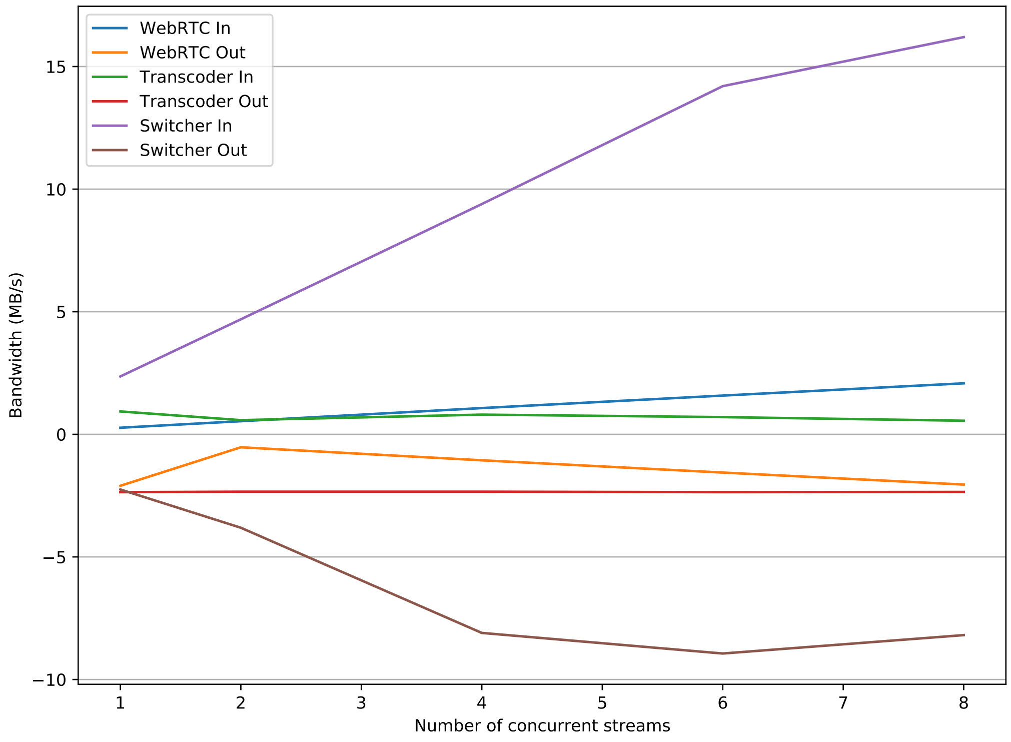

Figure 4.4: Pod bandwidth consumption

Figure 4.5 shows the aggregate consumed bandwidth for each worker node. We can observe a mirrored symmetry on the abscissa axis, denote the same level of inbound and outbound traffic. Desperados is the node which requires higher bandwidth availability, largely surpassing Imperial's requirements. This happens due all the video processing are being performed at the edge.

4.3 Summary

Experimental validation environment was detailed, the machines configuration presented and the cluster configuration was described in a step by step fashion.

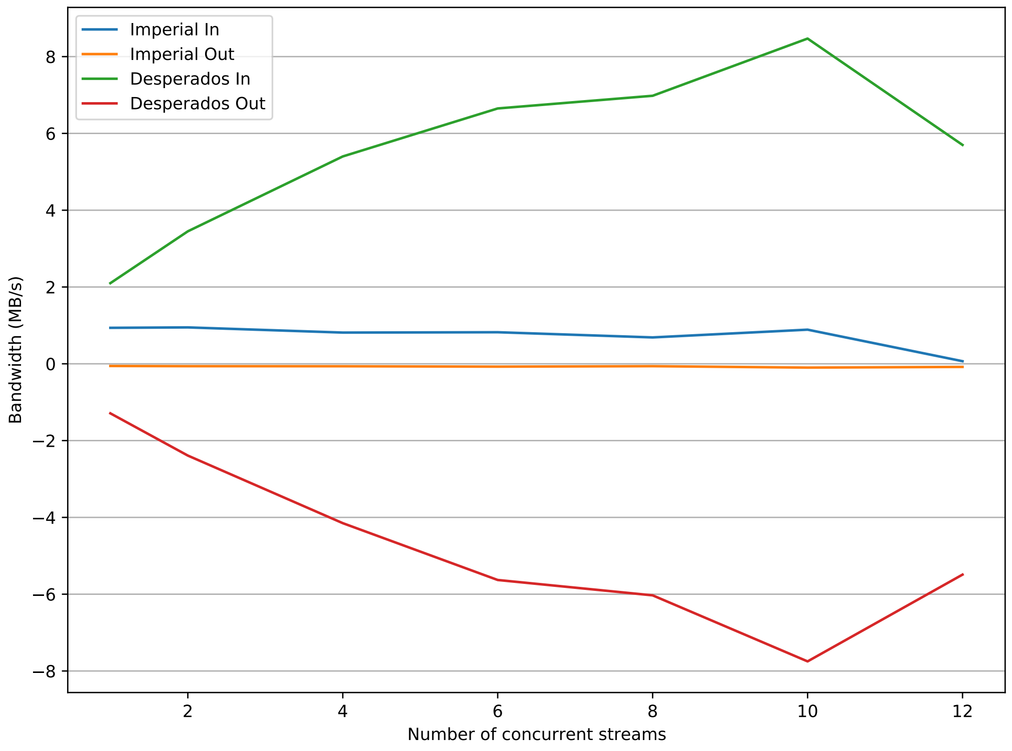

Figure 4.5: Node bandwidth consumption

Afterwards, the three contemplated scenarios were analysed and its results discussed, a LAN environment, a traditional central cloud setup and an Edge powered configuration. For each of them end to end stream latency, nodes bandwidth and individual pods bandwidth consumption were measured. End to end latency, arguably the most important metric in a real-time system was similar for fewer streams but its difference widthened with the increase of concurrent streams. Apart from the LAN configuration, a unlikely scenario to be found in the wild, the Edge configuration emerged as the fastest, largely surpassing the Central Cloud approach.

The bandwidth measurements also benefit the edge scenario in detriment of the more traditional approach, as all the heavier deployment tasks are handled at Desperados.

All in all, our implementation behaves as indented and our hypothesis was validated, Realtime livestreaming platform greatly benefits from an Edge Cloud architecture.

Chapter 5

Conclusions and Future Work

In this chapter conclusions are made, concerning the implemented architecture, additionally future work is also discussed.

These days, the growth in mobile devices number combined with constant internet connectivity fostered Crowdsourced Video Streaming. Live streams often need to be transcoded so they have the correct encoding, that is, appropriate for the final media platform. These are many times destined to social network platforms, which use are built on top of a centralised cloud architecture.

With this two problems arise, trust issues in social network platforms combined with the high latency and low bandwidth usual of their centralised data centre design. These problems were mitigated by developing an architecture witch supports a decentralised Crowdsourced Live Journalism platform.

5.1 Goals Fulfilment

The proposed goals were achieved as the state of the art focusing on the key areas an topics to implement the envisioned architecture was successfully performed as well as the definition and implementation of a scalable architecture adequate for a Crowdsourced Live Journalism platform was attained.

Mogplay was improved, porting the existing solution — a container based infrastructure — into an orchestrated infrastructure managed by Kubernetes. Said infrastructure is easily configurable through deployment template configuration files and highly scalable. Moreover, monitoring mechanisms, Prometheus and Grafana were introduced, to assess the system's overall performance, enabling us to have a better overall idea of its resource footprint. It also helped us identify some anomalous behaviours.

Finally, a non trivial fallback mechanism was developed. The live stream, upon its transcoder hard failure, is successfully recovered in a transparent manner for the end-user, resulting in remarkably better user experience, than what was previously in place.

Validation tests were performed, resulting in encouraging results for our edge cloud approach, as it presented lower delay than a solely cloud alternative. Expended bandwidth also concentrates on the edge node, what means internet bandwidth consumption to the central data centre is minimal.

5.2 Future Work

Concerning future work, more components of the system could be replicated, thus enabling an even higher degree of horizontal application scaling. This would improve the current prototype, allowing for colossal event coverage and possibly increasing the supported resolutions and video encodings.

Additionally, performing a larger scale stress test to assess which components are bottlenecking the system would be beneficial. Most likely the WebRTC Server and the Video Switcher, as they are the services were all the streams converge.

Moreover, determining to which transcoder a new stream should be allocated to could be largely improved, taking into account a transcoder's current utilisation or even the mobile device's capabilities by assigning a more powerful transcoder to higher-end devices. Also, stream scaling decision enhancements would greatly contribute to better resource optimisation.

Finally, extra versions of the broadcast stream could be transcoded to popular resolutions and bitrates, this would mean an end viewer would be able to enjoy the broadcast in the best possible bitrate for its device and network constraints. If enough resources are available the four previewed streams could receive the same treatment, therefore even if the outputted stream were to be switched, multiple versions would already be available.

Appendix A

Validation Data

In this chapter, the source data in which graphs displayed in Chapter 4 are based, can be consulted bellow.

Table A.1: Nodes Bandwidth Consumed (MB/s)

Streams	Imperial		Desperados	
	In (+)	Out (-)	In (+)	Out (-)
1	0.937	0.057	2.1	1.29
2	0.948	0.062	3.45	2.39
4	0.813	0.064	5.4	4.15
6	0.822	0.074	6.65	5.63
8	0.687	0.062	6.98	6.03
10	0.889	0.099	8.47	7.75
12	0.0675	0.081	5.7	5.49

Table A.2: Pods Bandwidth Consumed (MB/s)

Streams	WebRTC Server		Transcoder (Active)		Switcher	
	In (+)	Out (-)	In (+)	Out (-)	In (+)	Out (-)
1	0.268	2.1	0.933	2.36	2.36	2.25
2	0.535	0.529	0.578	2.34	4.69	3.81
4	1.07	1.06	0.803	2.34	9.39	8.1
6	1.58	1.56	0.702	2.36	14.2	8.94
8	2.08	2.05	0.554	2.35	16.2	8.19

Table A.3: End to End delay LAN (s)

Streams		Delay (s)	Average
1	1	1.21	1.210
2	1	1.95	1.940
	2	1.93	
4	1	1.97	1.975
	2	1.98	
	3	1.99	
	4	1.96	
6	1	1.96	1.973
	2	1.98	
	3	1.99	
	4	1.96	
8	1	5.02	5.028
	2	4.09	
	3	5.09	
	4	5.91	
10	1	7.83	7.408
	2	7.90	
	3	6.98	
	4	6.92	
12	1	11.07	9.990
	2	10.95	
	3	8.01	
	4	9.93	

Table A.4: End to End delay Central Cloud (s)

Streams		Delay (s)	Average
1	1	1.91	1.910
2	1	1.96	1.950
	2	1.94	
4	1	1.23	1.408
	2	1.21	
	3	1.99	
	4	1.20	
6	1	6.00	5.835
	2	5.17	
	3	6.17	
	4	6.00	
8	1	9.90	9.220
	2	8.95	
	3	10.04	
	4	7.99	
10	1	14.14	14.770
	2	14.96	
	3	15.01	
	4	14.97	
12	1	19.04	18.295
	2	17.02	
	3	18.00	
	4	19.12	

Table A.5: End to End delay Edge (s)

Streams		Delay (s)	Average
1	1	1.16	1.160
2	1	1.92	1.920
	2	1.92	
4	1	1.18	1.193
	2	1.20	
	3	1.18	
	4	1.21	
6	1	6.91	6.478
	2	6.04	
	3	6.01	
	4	6.95	
8	1	8.04	8.233
	2	7.95	
	3	8.88	
	4	8.06	
10	1	9.87	9.168
	2	8.04	
	3	8.87	
	4	9.89	
12	1	12.91	12.108
	2	11.78	
	3	11.94	
	4	11.80	

Table A.6: Pods CPU (cores) and RAM (MiB)

Streams	Janus		Transcoder (one active)		Switcher	
	CPU	RAM	CPU	RAM	CPU	RAM
0	0.005	19.570	0.000	54.450	0.000	4.390
1	0.020	25.980	0.690	153.140	0.112	12.000
2	0.050	24.000	0.800	147.000	0.150	12.000
4	0.090	25.000	0.700	150.000	0.240	12.000
6	0.135	31.670	0.700	149.000	0.404	15.250
8	0.155	34.370	0.646	114.130	0.400	15.810
10	0.187	39.130	0.672	161.480	0.313	16.640
12	0.210	41.150	0.664	196.950	0.262	17.390

References

[1] Tanja Aitamurto. Crowdsourcing in journalism. https://oxfordre.com/communication/view/10.1093/acrefore/9780190228613.001.0001/acrefore-9780190228613-e-795, 04 2019. DOI: 10.1093/acrefore/9780190228613.013.795 [Online - Last access, 26 July 2020].

[2] N. Alshuqayran, N. Ali, and R. Evans. Towards micro service architecture recovery: An empirical study. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 47–4709, 2018. DOI: 10.1109/ICSA.2018.00014.

[3] Kashif Bilal and Aiman Erbad. Edge computing for interactive media and video streaming. *2017 2nd International Conference on Fog and Mobile Edge Computing, FMEC 2017*, pages 68–73, 2017. DOI: 10.1109/FMEC.2017.7946410.

[4] Kashif Bilal, Aiman Erbad, and Mohamed Hefeeda. Crowdsourced multi-view live video streaming using cloud computing. *IEEE Access*, 5:12635–12647, 2017. DOI: 10.1109/ACCESS.2017.2720189.

[5] Alfred Bratterud, Alf Andre Walla, Harek Haugerud, Paal E. Engelstad, and Kyrre Begnum. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *Proceedings - IEEE 7th International Conference on Cloud Computing Technology and Science, CloudCom 2015*, pages 250–257. Institute of Electrical and Electronics Engineers Inc., 2016. DOI: 10.1109/CloudCom.2015.89.

[6] S. Hakansson C. Holmberg and G. Eriksson. Web real-time communication use cases and requirements. RFC 7478, RFC Editor, 3 2015.

[7] Cisco. Cisco annual internet report (2018–2023). https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf, 2018. [Online - Last access 9 June 2020].

[8] Cisco. 2020 global networking trends report. https://www.cisco.com/c/m/en_us/solutions/enterprise-networks/networking-report.html, 2020. [Online - Last access 9 June 2020].

[9] Docker. https://www.docker.com/. [Online - Last access, 21 January 2020].

[10] Docker networks. https://docs.docker.com/network/. [Online - Last access, 30 June 2020].

[11] Alexis Ducastel. Benchmark results of kubernetes network plugins (cni) over 10gbit/s network (updated: April 2019). https://itnext.io/benchmark-results-of-kubernetes-network-plugins-cni-over-10gbit-s-network-updated-april-2019-4a9886efe9c4, 2019. [Online - Last access 1 July 2020].

[12] Justin Ellingwood. Comparing kubernetes cni providers: Flannel, calico, canal, and weave. https://rancher.com/blog/2019/2019-03-21-comparing-kubernetes-cni-providers-flannel-calico-canal-and-weave/, 2019. [Online - Last access 1 July 2020].

[13] Martin Fowler and James Lewis. Microservices - a definition of this new architectural term. https://martinfowler.com/articles/microservices.html, 2014. [Online - Last access 26 June 2020].

[14] Nikita V. Ghodpage and R. V. Mante. Privacy Preserving and Information Sharing in Decentralized Online Social Network. *Proceedings of the International Conference on Inventive Communication and Computational Technologies, ICICCT 2018*, pages 152–155, 2018. DOI: 10.1109/ICICCT.2018.8473268.

[15] P. Heidari, Y. Lemieux, and A. Shami. Qos assurance with light virtualization - a survey. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 558–563, 2016. DOI: 10.1109/CloudCom.2016.0097.

[16] N. Kapočius. Performance Studies of Kubernetes Network Solutions. In *2020 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)*, pages 1–6. IEEE, apr 2020. DOI: 10.1109/eStream50540.2020.9108894.

[17] Kubernetes Pod documentation. https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/. [Online - Last access, 09 February 2020].

[18] Jianjun Li, Wei Li, and Ming Li. Towards an Out-of-the-Box Cloud Application Monitoring Framework. *Proceedings - 3rd IEEE International Conference on Cyber Security and Cloud Computing, CSCloud 2016 and 2nd IEEE International Conference of Scalable and Smart Cloud, SSC 2016*, pages 46–49, 2016. DOI: 10.1109/CSCloud.2016.39.

[19] Filipe Manco, Jose Mendes, Kenichi Yasukata, Costin Lupu, Simon Kuenzer, Costin Raiciu, Florian Schmidt, Sumit Sati, and Felipe Huici. My VM is Lighter (and Safer) than your Container. *SOSP 2017 - Proceedings of the 26th ACM Symposium on Operating Systems Principles*, pages 218–233, 2017. DOI: 10.1145/3132747.3132763.

[20] Peter Mell and Timothy Grance. The nist definition of cloud computing. Technical report, National Institute of Standards and Technology - US Department of Comemerce, 2011. DOI: 10.6028/NIST.SP.800-145.

[21] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment, 2014. [Online - Last access 21 January 2020].

[22] R. Morabito, J. Kjällman, and M. Komu. Hypervisors vs. lightweight virtualization: A performance comparison. In *2015 IEEE International Conference on Cloud Engineering*, pages 386–393, March 2015. DOI: 10.1109/IC2E.2015.74.

[23] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: a transport protocol for real-time applications. RFC 3350, RFC Editor, 7 2003.

[24] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, Oct 2016. DOI: 10.1109/JIOT.2016.2579198.

[25] Rui Shu, Peipei Wang, Sigmund A Gorski III, Benjamin Andow, Adwait Nadkarni, Luke Deshotels, Jason Gionta, William Enck, and Xiaohui Gu. A study of security isolation techniques. *ACM Computing Surveys (CSUR)*, 49(3):50, 2016. DOI: 10.1145/2988545.

[26] Salman Taherizadeh, Ian Taylor, Andrew Jones, Zhiming Zhao, and Vlado Stankovski. A network edge monitoring approach for real-time data streaming applications. In José Ángel Bañares, Konstantinos Tserpes, and Jörn Altmann, editors, *Economics of Grids, Clouds, Systems, and Services*, pages 293–303, Cham, 2017. Springer International Publishing. DOI: 10.1007/978-3-319-61920-0_21.

[27] A. Tosatto, P. Ruiu, and A. Attanasio. Container-based orchestration in cloud: State of the art and challenges. In *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 70–75, July 2015. DOI: 10.1109/CISIS.2015.35.

[28] Martín Varela, Lea Skorin-Kapov, and Touradj Ebrahimi. *Quality of Service Versus Quality of Experience*, pages 85–96. Springer International Publishing, Cham, 2014. DOI: 10.1007/978-3-319-02681-7_6.