POLITECNICO DI TORINO

master’s Degree in Computer Engineering

master’s Degree Thesis

Cloudifying desktop applications with Kubernetes

Supervisor:
Prof. Fulvio Risso

Candidate:
Antonio Riccardo Roccaro

Academic Year 2019/2020

Table of contents

Chapter 1 - Introduction

The aim of this thesis is to explain how to offload a desktop application execution to a remote Kubernetes cluster node by maintaining its control on the local host through the Graphical User Interface and the audio stream remotization.

Currently, it is often necessary to execute desktop applications requiring more system resources (in terms of CPU, GPU, RAM, network et cetera) than the user's machines may provide. This lack of resources leads to a significant worsening of the application user experience or the impossibility of using the application itself. In this regard, the project described in this thesis enables to offload the desktop application execution to a remote machine that has sufficient resources to execute it (by maintaining the application control on the local host).

In effect, thinking about the client-server paradigm, the execution of an application on a remote machine is not a new concept. However, that kind of application has not a desktop Graphical User Interface, but it is provided with a web user interface at most. Therefore, in order to implement the architecture described in this thesis, the first challenge is running a desktop application on a remote machine and the second one is making the Graphical User Interface available on the host from which the application will be launched.

In order to achieve these goals, considered the significant potential of the application containerization technologies and of the container execution on clusters, it has been chosen to containerize some sample applications by using Docker and then executing the obtained containers on a Kubernetes cluster.

Obviously, since this project deals with desktop applications, their mere containerization will be not sufficient as it is necessary to interact somehow with the remote application through its own User Interface from the local host. Consequently, it has been chosen the VNC protocol to remotize the Graphical User Interface and the PulseAudio tool to achieve the same goal for the audio reproduction. Moreover, to reinforce the architecture from a security point of view, the SSH protocol has been used to create encrypted tunnels wrapping the audio and the video streams.

The base architecture might work by using the protocols and tools described above. Nevertheless, the user experience may not result particularly good. In fact, several other features may be implemented, such as the data persistency.

The mere containerization and on-cluster execution do not guarantee that the data obtained from the running application will persist once the container execution ends. In fact, for instance, if the application configuration is modified through its settings menu by the user, once the container terminates its execution, all the changes will be lost. This is an undesirable behavior in terms of user experience. Therefore, it has been chosen to use the Kubernetes Persistent Volume Claims to solve this problem.

So far, the focus was put mostly on the server-side challenges but there are also other ones to face on the client-side as well. In fact, since on the remote container VNC and PulseAudio will be respectively employed to manage the video and audio streams, the installation of a proper software on the local machine is required to interact with those tools.

In this case, different solutions have been found: the simplest one is natively installing the required software on the local machine. However, as shown in the next chapters, this solution is not optimal since it causes trouble in terms of user experience. In fact, the software installation may cause configuration and execution conflicts with the pre-existent applications. In addition to this, further user's efforts are necessary to install the required software consisting in several applications and tools.

Consequently, a possible solution may be the exploitation of the container features on the client-side as well. Thus, since the container is indeed isolated from the host executing it and since it contains all the required applications and tools, the final user needs to install and configure exclusively the Docker engine.

Obviously, this solution also creates some challenges. In fact, since the container is isolated from the host, it will be necessary to find a way to bind the container's audio and video interfaces to those of the hosts.

The previous client-side implementation solutions carry one more challenge due to the fact that in both cases it will be required to find a way to make the client-side interact with the remote container in order to create the audio/video tunnels (either encrypted or clear). This aspect as well as the Kubernetes NodePort services used for this purpose will be discussed further.

Finally, assuming that the local host could be part of a Kubernetes cluster as well, it has been decided to exploit some Kubernetes features to implement a *fully cloudified* environment. Including in this case, there will be other problems to face. In fact, since Kubernetes automatically schedules the pods among its nodes, it will be required to force the scheduling in order to execute respectively the server-side on a remote node and the client-side on the local node. In this *fully cloudified* implementation, as it will be explained further, unlike the *native* and the *containerized* implementations, the container reachability will be facilitated due to some Kubernetes features exploitation.

In the next chapter an overview about the existing projects trying to face problems similar to that aimed to be resolved in this thesis will be presented.

Chapter three will deal with the theoretical concepts behind the two widely used technologies, Docker and Kubernetes. Particularly, great emphasis will be placed on Docker images, on their creation through the *Dockerfiles* definition and their execution as Docker containers. Then, the Kubernetes *Pods* concept will be explored, as well as their creation through the *Deployments* and the *Jobs*, and their reachability through the *Services*. Kubernetes core objects (*Pods*, *Deployments*, *Jobs* and *Services*), will be investigated in detail by describing their structure and usage. Successively, an overview of *Secrets* (used to share SSH keys with the *Pods*), *Persistent Volume Claims* (used for data persistency) and *Node Affinity* (used for the *Pods* scheduling) will be provided.

Chapter four shows the general architecture aimed to be built. This part illustrates how to manage the audio and the video streams and how it is necessary to schedule each component.

In chapter five the server-side implementation of the architecture illustrated in the chapter four will be discussed. Firstly, a description of the tools used to create the encrypted tunnels wrapping the audio and the video streams will be provided. Successively, the focus will be placed on the audio and the video management, the security issues, the pods scheduling (along with their reachability from the inside and the outside of the cluster), the data persistency and the application concurrency.

Chapter six revolves around the client-side implementation details based on the three execution modes (native, Docker container, Kubernetes pod) and the explanation of the client-side aspects corresponding to those described in the server-side. In particular, the audio and video management in native mode, as well as the remote pod reachability from the client-side, will be discussed. The aforementioned audio/video management and pod reachability will be explored for the Docker mode as well and it will be explained how the Docker interacts with the host for the audio/video reproduction. Finally, the *fully cloudified* implementation will be discussed. It shares some aspects with the Docker container implementation, but it has additional features (i.e. an easier pod reachability system compared to the Docker implementation one).

Chapter seven is about the project validation. It will be analyzed the resources consumption during the project execution and the remote application startup timing. The results obtained by the client-side execution in each mode (native, Docker container and Kubernetes pod) will be compared to each other and then to those related to the execution of the offloaded application natively installed on the host. The applications currently supported in this project will be illustrated and it will be explained that it is possible to execute the project both in a vanilla Kubernetes installation and in a cluster running the Liqo project, which virtualizes an entire remote Kubernetes cluster as a local node.

Chapter eight presents the conclusions of this thesis and possible future project improvements.

Chapter 2 - State of art

2.1. Introduction

The application execution offloading on a remote host is a feature very useful in case the local machine has not sufficient resources to execute certain application. For this reason, there are some solutions to solve this problem, or similar ones, even if they do not precisely produce the effect this thesis aims at. The three solutions described afterwards are: Virtual Desktop, Desktop as a Service and Android devices apps usage from PC.

2.2. Virtual Desktop

Virtual desktops are preconfigured images of operating systems and applications in which the desktop environment is separated from the physical device used to access it[1]. Users can access their virtual desktops remotely over a network. Any endpoint device, such as a laptop, smartphone or tablet, can be used to access a virtual desktop. The virtual desktop provider installs client software on the endpoint device, and the user then interacts with that software on the device.

A virtual desktop looks and feels like a physical workstation. The user experience is often even better than a physical workstation because powerful resources, such as storage and back-end databases, are readily available. Users may or may not be able to save changes or permanently install applications, depending on how the virtual desktop is configured. Users experience their desktop exactly the same way every time they log in, in spite of the device from which they are logging into.

There are a few different types of virtual desktops and desktop virtualization technologies:

- With *host-based virtual machines*, one virtual machine is allocated to each individual user at login. With persistent desktop technology, that user connects to the same VM each time they log in, which allows for desktop personalization. Host-based machines can also be physical machines hosting an operating system that remote users log into.
- A virtual machine can also be *client-based*, where the operating system is executed locally on the endpoint. The advantage of this type of virtual desktop is that a network connection is not required for the user to access the desktop.
- *Virtual desktop infrastructure* (VDI) refers to a type of desktop virtualization that allows desktop workstation or server operating systems to run on virtual machines that are hosted on a hypervisor in on-premises servers. The user experiences the operating system and applications on an endpoint device, just as if they were running locally.
- With *desktops as a service* (DaaS), a service provider hosts VDI workloads out of the cloud and provides apps and support for enterprise users.

[1] The content of this section has been taken, with some adaptations, from: Vmware official website - Virtual Desktop, https://www.vmware.com/topics/glossary/content/virtual-desktops, October 10th, 2020.

Virtual desktop providers abstract the operating system from a computer's hardware with virtualization software. Instead of running on the hardware, the operating system, applications and data run on a virtual machine. An organization may host the virtual machine on premises. It is also common to run a virtual desktop on cloud-based virtual machines. Previously, only one user could access a virtual desktop from a single operating system. The technology has evolved to allow many users to share an operating system that is running multiple desktops.

IT administrators can choose to purchase virtual desktop thin clients for their VDI, or repurpose older or even obsolete PCs by using them as virtual desktop endpoints, which can save money. However, any money saved on physical infrastructure costs may need to be quickly reallocated to software licensing fees for virtual desktops.

A virtual desktop infrastructure provides the option for users to bring their own device, which can again save IT departments money. This flexibility makes virtual desktops ideal for seasonal work or organizations that employ contractors for temporary work on big projects. Virtual desktops also work well for salespeople who travel frequently because their desktop is the same and they have access to all the same files and applications in spite of the location from where they are working.

There are many advantages regarding the use of a virtual desktop environment:

- *Security*: One way in which virtual desktops can be superior to physical desktop machines is security. Data is stored in the data center and not on individual endpoint machines, which can allow for greater data security. If an endpoint device is stolen, it does not contain any data for thieves to access.
- *Flexibility*: For organizations with a flexible workforce, virtual desktops have a clear advantage. IT administrators can quickly and easily allocate virtual desktops without the need to provision expensive physical machines to users who might only need them for a short time.
- *Cost*: Because virtual desktops require less physical equipment and maintenance, they can be more cost-effective than physical desktops.
- *Easy management*: An IT department can easily manage a large number of far-flung virtual desktops from a central location. Software updates are faster and easier because they can be done all at once instead of machine by machine.
- *Computing power*: Thin clients are all that are needed for virtual desktops because the computing power for the desktops is coming from a powerful data center.

There also are downsides when using a virtual desktop environment:

- If the data center runs out of storage space, users are not able to access their desktops.
- Large storage environments that have the capacity to store data for multiple virtual desktops can be expensive.
- Poor network connectivity will adversely affect the user experience, and users will also not be able to access their desktop if there is no network connection (this may be the most significant disadvantage of using a virtual desktop).
- Depending on how an organization runs, the benefits of virtual desktops often still exceed the potential challenges.

A virtual desktop allows users to access their desktop and applications from anywhere on any kind of endpoint device, while IT organizations can deploy and manage these desktops from a centrally located data center. Many organizations move to a virtual desktop environment because virtual desktops are usually centrally managed, which eliminates the need for updates and app installations on individual machines. Also, endpoint machines can be less powerful, since most computing happens in the data center.

Virtual desktops are as easy to use as physical desktops. Users simply log in to their desktop from their chosen device and connect via the network to a remotely located virtual machine that presents the desktop on the endpoint device. Users can interact with applications on a virtual desktop in the same way that they would on a physical desktop. Users may or may not be able to personalize or save data locally on a virtual desktop, depending on which desktop virtualization technology they are using.

Virtual Desktop Infrastructure or VDI is the desktop virtualization environment that deliver virtual desktops to endpoint devices from a data center located on premises or in the cloud. The operating system for the virtual desktop lives in the data center, not the endpoint. In most cases, these operating systems and computing resources run on virtual machines (VMs) hosted by hypervisors rather than on physical machines.

2.3. Desktop as a Service

Desktops-as-a-Service or simply DaaS, securely delivers virtual apps and desktops from the cloud to any device or location[2]. This desktop virtualization solution provisions secure SaaS and legacy applications as well as full Windows-based virtual desktops and delivers them to your workforce. DaaS offers a simple and predictable pay-as-a-go subscription model, making it easy to scale up or down on-demand. This turnkey service is easy to manage, simplifying many of the IT admin tasks of desktop solutions.

Desktops as a Service (DaaS) delivers virtual applications and desktop services via a public or private cloud service. This service can be accessed through an internet connection via an html-based web browser or a secure application downloaded to a device such as a laptop, desktop, thin client or tablet.

DaaS is offered as a subscription service and is multitenant in nature. The backend virtual desktop infrastructure (VDI) infrastructure, including the virtual machines that run desktop operating systems, is hosted by a third-party cloud provider. The DaaS provider then streams the virtual desktops to a customer's end-user devices.

DaaS providers manage the VDI deployment, as well as the maintenance, security, upgrades, data backup, and storage. And the customer manages the applications and desktop images. DaaS is a good choice for organizations that does not want to invest in and manage their own on-premises VDI solution.

DaaS is a form of Virtual Desktop Infrastructure (VDI), hosted in the cloud. With VDI, an organization deploys virtual desktops from its own on-premises data centers. In-house IT

[2] The content of this section has been taken, with some adaptations, from: Citrix official website - What is Desktop as a Service (DaaS)?, https://www.citrix.com/en-gb/glossary/what-is-desktop-as-a-service-daas.html, October 10th, 2020.

teams are responsible for deploying the virtual desktops as well as purchasing, managing, and upgrading the infrastructure.

DaaS is essentially the same thing but the infrastructure is cloud-based. Organizations that subscribe to a DaaS solution does not need to manage their own hardware.

Desktop as a Service has several pros:

- Flexibility: Employees, seasonal workers and contractors can securely access their applications, remote desktops, and data from anywhere on cost effective devices, keeping them productive, no matter where they work.
- Scalability: Rapidly scale up applications and desktops when needed and then scale down when you no longer need them, keeping IT costs inline.
- Business continuity: DaaS can be a simple way to support a disaster recovery (DR) plan.
- Cost savings: Pay for only what you use.
- Security: DaaS provides a secure access point for users and simplifies desktop and app management processes and procedures. With access to applications and desktops in the cloud, data is securely stored and protected against data loss or device theft.

2.4. Use apps from your Android device on your PC

Microsoft Windows, with *Your Phone apps*, offers the possibility of instantly access the Android apps installed on the user's mobile device right on user's PC[3]. Using a Wi-Fi connection, Apps allows user to browse, play, order, chat, and more – all while using user PC's larger screen and keyboard. The user can add his Android apps as favorites on his PC, pin them to his Start menu and taskbar, and open them in separate windows to use side-by-side with apps on his PC – helping the user stay productive.

To interact with the Android apps, user can use his PC's mouse, trackpad, keyboard, pen or touch-enabled screen to open, type, scroll, and interact with apps. A few tips for using user's mouse and keyboard:

- Single click will behave the same as any single touch/tap interaction.
- Right click anywhere on user's phone screen to navigate to the previous page.
- Click and hold will behave the same as a tap/hold interaction.
- Click and hold and drag to select content.
- Mouse scroll to move between pages vertically or horizontally.

Some games and apps might not work with a mouse or keyboard. User will need to use a touch-enabled PC to interact with them.

Apps the user opens on his PC will be running from his Android device. User's Android device needs to be on and connected to the same Wi-Fi network in order for this to work. The Your Phone app is connecting and mirroring apps to user's PC from user's Android phone.

[3] The content of this section has been taken, with some adaptations, from: Microsoft official website - Use apps from your Android device on your PC, https://support.microsoft.com/en-us/help/4577326/use-apps-from-your-android-device-on-your-pc, October 10th, 2020.

User needs to connect his Android device to his PC via the Your Phone app in order to use this experience.

Finally, user can only open one Android app at a time. The Your Phone app is mirroring his Android device’s screen and the opened app in it. If the user opens a new app, the one already opened will be replaced with the new app.

Chapter 3 - Used technologies

3.1. Introduction

In this thesis the aim is to offload the application execution to a remote machine and to maintain its control on the local host. Since we cannot expect to install each desired software on the first one nor to choose by hands in which one that software has to be executed, we need to find a way to automatically schedule the application execution without installing it on the remote hosts.

The first challenge is avoiding a lot of software installations in the remote machines. Here it comes the application containerization concept. «A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another»[4]. Once a container has been created, it will be executed through a container runtime engine installed in the machine. So, we just need to install it once and then run each desired container without installing the related software natively on the host. Moreover, this way we can also avoid execution and configuration conflicts. There are several container engines, we decided to use Docker.

The second challenge is to automatically schedule the container execution on a remote machine. In fact, to achieve the application offloading, containerizing is not sufficient because we actually need to make it run on another host that belongs to a cluster. As said before, since the aim is not choosing by hand the remote machine that will execute that software, an orchestrator is required. There are several orchestrators, but it was chosen to use Kubernetes.

3.2. Docker

3.2.1. Introduction

Docker is an open platform for developing, shipping, and running applications[5]. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production. Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host. Containers are lightweight because they do not need the extra load of a hypervisor, but run directly within the host machine's kernel. This means you can run more containers on a given hardware combination than if you were using

[4] Docker official website - What is a container, https://www.docker.com/resources/what-container, September 30th, 2020.

[5] The content of this section has been taken, with some adaptations, from: Docker official website - Docker overview, https://docs.docker.com/get-started/overview/, September 30th, 2020; Docker official website - What is a container, https://www.docker.com/resources/what-container, September 30th, 2020; Docker official website - Docker overview, https://docs.docker.com/get-started/overview/, September 30th, 2020.

virtual machines. You can even run Docker containers within host machines that are actually virtual machines.

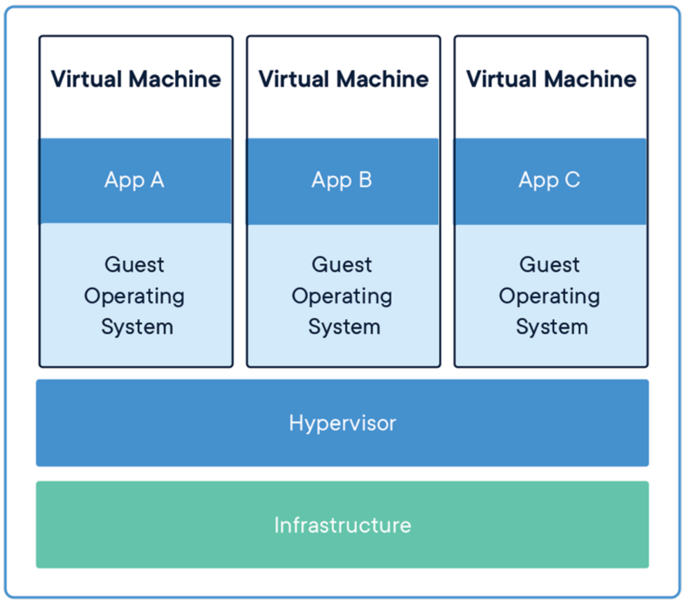

Figure 1: Virtual machine model[6]

The picture above shows the virtual machine model. As you can see, virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.

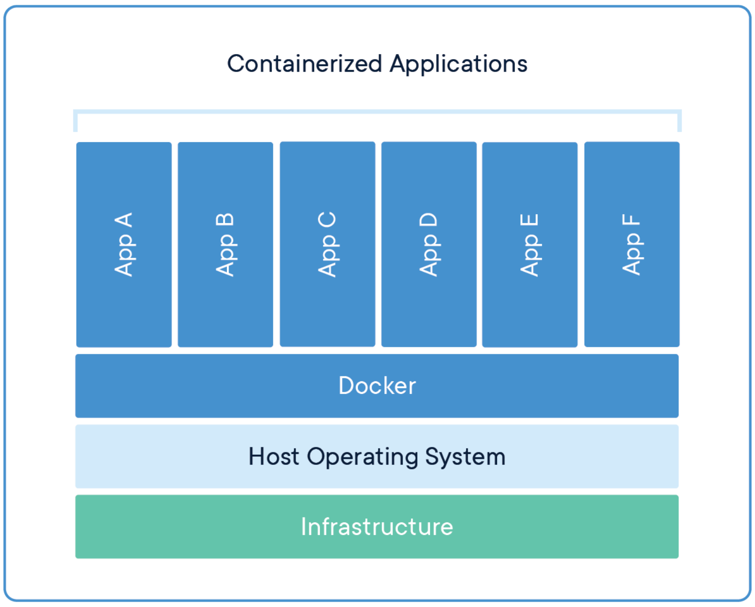

Figure 2: Docker container model[7]

[6] Docker official website - What is a container, https://www.docker.com/resources/what-container, September 30th, 2020.
[7] Idem.

The picture above shows the container model. As you can see, unlike the VMs, containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems.

As said before, containers and virtual machines can be used together to provide a great deal of flexibility in deploying and managing app.

In the previous section we mentioned that a container will be executed by a container runtime engine. For this purpose, Docker provides the so called "Docker Engine" that is a client-server application with these major components:

- A server, which is a type of long-running program called a daemon process (the *dockerd* command).
- A REST API, which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
- A command line interface (CLI) client (the *docker* command).

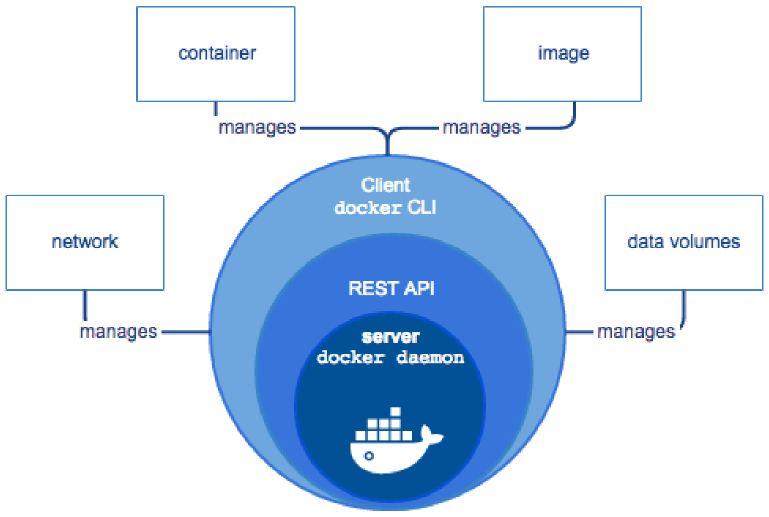

Figure 3: Docker engine[8]

The CLI uses the Docker REST API to control or interact with the Docker daemon through scripting or direct CLI commands. Many other Docker applications use the underlying API and CLI.

The daemon creates and manages Docker objects, such as images, containers, networks, and volumes.

[8] Docker official website - Docker overview, https://docs.docker.com/get-started/overview/, September 30th, 2020.

3.2.2. Docker images

A Docker *image* is a read-only template with instructions for creating a Docker container[9]. Often, an image is *based on* another image, with some additional customization. For example, you may build an image which is based on the *ubuntu* image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a *Dockerfile* with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

The command that creates the image starting from the Docker file is the following one:

docker build [OPTIONS] PATH | URL | -

3.2.3. Dockerfiles

Thus, a Docker image consists of read-only layers each of which represents a Dockerfile instruction. The layers are stacked and each one is a delta of the changes from the previous layer[10]. Consider the following Dockerfile example:

```
ARG UBUNTU_VERSION=18.04
FROM ubuntu:${UBUNTU_VERSION}

LABEL MAINTAINER=”Antonio Riccardo Roccaro”

SHELL ["/bin/bash", "-c"]

ARG APPLICATION=firefox

ENV SSH_PORT=22

RUN apt-get install -y \
        $APPLICATION \
        openssh-server

EXPOSE $SSH_PORT

ADD myapp.sh /image/destination/path/
COPY [ "host_file1", "host_file2", "/image/destination/path/" ]

ENTRYPOINT [ "/image/destination/path/myapp.sh" ]
```

Each instruction creates one layer:

[9] The content of this section has been taken, with some adaptations, from: Docker official website - Docker overview, https://docs.docker.com/get-started/overview/, October 1st, 2020.

[10] The content of this section has been taken, with some adaptations, from: Docker official website - Best practices for writing Dockerfiles, https://docs.docker.com/develop/develop-images/dockerfile_best-practices/, October 1st, 2020; Docker official website – Dockerfile reference, https://docs.docker.com/engine/reference/builder/, October 2nd, 2020.

- FROM creates a layer from the ubuntu:18.04 Docker image. The FROM instruction initializes a new build stage and sets the Base Image for subsequent instructions. As such, a valid Dockerfile must start with a FROM instruction. The image can be any valid image – it is especially easy to start by pulling an image from the Public Repositories. ARG is the only instruction that may precede FROM in the Dockerfile. In the example above the *UBUNTU_VERSION* ARG is used to specify which ubuntu version to use as base image. Since that value is set by default to *18.04*, that one will be used if no proper different *--build-arg* parameter will be specified during build time. In fact, to specify a different version, i.e. the *20.04*, you need to run the *build* command as follows:

  docker build --build-arg UBUNTU_VERSION=20.04 .

  Note that an ARG declared before a FROM is outside of a build stage, so it can't be used in any instruction after a FROM;
- ARG defines a variable that users can pass at build-time to the builder with the *docker build* command using the *--build-arg <varname>=<value>* flag. If a user specifies a build argument that was not defined in the Dockerfile, the build outputs a warning. A Dockerfile may include one or more ARG instructions. An ARG instruction can optionally include a default value like in the example above in which the *UBUNTU_VERSION* argument has the *18.04* default value. Note that the ARG value is not persisted in the final image, use the ENV instruction instead;
- ENV sets the environment variable <key> to the value <value>. This value will be in the environment for all subsequent instructions in the build stage and can be replaced inline in many as well. The value will be interpreted for other environment variables, so quote characters will be removed if they are not escaped. Like command line parsing, quotes and backslashes can be used to include spaces within values. The environment variables set using ENV will persist when a container is run from the resulting image. You can view the values using *docker inspect*, and change them using *docker run --env <key>=<value>*. As for the RUN instruction, it is possible to split the env variable declaration in few lines (one for each) by using the \ (backslash);
- RUN will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile. You can use a \ (backslash) to continue a single RUN instruction onto the next line as shown in the example above;
- SHELL allows the default shell used for the *shell* form of commands to be overridden. The default shell on Linux is ["/bin/sh", "-c"], and on Windows is ["cmd", "/S", "/C"]. The following instructions can be affected by the SHELL instruction when the *shell form* of them is used in a Dockerfile: RUN, CMD and ENTRYPOINT. Note that for *shell form* we refer to the one represented by ["something", "something", …, "something"] like the one used in the example above for the COPY and ENTRYPOINT instructions.

- LABEL adds metadata to an image. A LABEL is a key-value pair. To include spaces within a LABEL value, use quotes and backslashes as you would in command-line parsing. An image can have more than one label. You can specify multiple labels on a single line. As for the RUN instruction, it is possible to split the label declaration in few lines (one for each) by using the \ (backslash). Labels included in base or parent images (images in the FROM line) are inherited by your image. If a label already exists but with a different value, the most-recently-applied value overrides any previously-set value. To view an image's labels, use the *docker image inspect* command.
- EXPOSE informs Docker that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP, and the default is TCP if the protocol is not specified. The EXPOSE instruction does not actually publish the port. It functions as a type of documentation between the person who builds the image and the person who runs the container, about which ports are intended to be published. To actually publish the port when running the container, use the -p flag on *docker run* to publish and map one or more ports, or the -P flag to publish all exposed ports and map them to high-order ports. Regardless of the EXPOSE settings, you can override them at runtime by using the -p flag.
- ADD ["<src>",... "<dest>"] copies new files, directories or remote file URLs from <src> and adds them to the filesystem of the image at the path <dest>. Multiple <src> resources may be specified but if they are files or directories, their paths are interpreted as relative to the source of the context of the build. ADD obeys the following rules:
  - The <src> path must be inside the context of the build; you cannot *ADD ../something /something*, because the first step of a docker build is to send the context directory (and subdirectories) to the docker daemon;
  - If <src> is a URL and <dest> does not end with a trailing slash, then a file is downloaded from the URL and copied to <dest>;
  - If <src> is a directory, the entire contents of the directory are copied, including filesystem metadata. The directory itself is not copied, just its contents;
  - If <src> is a local tar archive in a recognized compression format (identity, gzip, bzip2 or xz) then it is unpacked as a directory;
  - If multiple <src> resources are specified, either directly or due to the use of a wildcard, then <dest> must be a directory, and it must end with a slash /;
  - If <dest> doesn't exist, it is created along with all missing directories in its path.
- COPY ["<src>",... "<dest>"] copies new files or directories from <src> and adds them to the filesystem of the container at the path <dest>. Multiple <src> resources may be specified but the paths of files and directories will be interpreted as relative to the source of the context of the build. COPY obeys the following rules:
  - The <src> path must be inside the context of the build; you cannot COPY *../something /something*, because the first step of a docker build is to send the context directory (and subdirectories) to the docker daemon;
  - If <src> is a directory, the entire contents of the directory are copied, including filesystem metadata;

- If multiple <src> resources are specified, either directly or due to the use of a wildcard, then <dest> must be a directory, and it must end with a slash /;
- If <dest> doesn't exist, it is created along with all missing directories in its path.

Note that although ADD and COPY are functionally similar, generally speaking, COPY is preferred. That's because it's more transparent than ADD. COPY only supports the basic copying of local files into the container, while ADD has some features (like local-only tar extraction and remote URL support) that are not immediately obvious. Consequently, the best use for ADD is local tar file auto-extraction into the image, as in ADD rootfs.tar.xz /;

- ENTRYPOINT ["executable", "param1", "param2"] allows you to configure a container that will run as an executable. Command line arguments to *docker run <image>* will be appended after all elements in an exec form ENTRYPOINT. This allows arguments to be passed to the entry point, i.e. *docker run <image> -d* will pass the *-d* argument to the entry point. You can override the ENTRYPOINT instruction using the *docker run --entrypoint flag*.

A part the Dockerfile instructions shown in the example above, there are the following others:

- CMD specifies the executable to execute when the image has been run. There can only be one CMD instruction in a Dockerfile. If you list more than one CMD then only the last CMD will take effect. The main purpose of a CMD is to provide defaults for an executing container. These defaults can include an executable, or they can omit the executable, in which case you must specify an ENTRYPOINT instruction as well. If CMD is used to provide default arguments for the ENTRYPOINT instruction, both the CMD and ENTRYPOINT instructions should be specified with the JSON array format. Both CMD and ENTRYPOINT instructions define what command gets executed when running a container. There are few rules that describe their co-operation:
  - Dockerfile should specify at least one of CMD or ENTRYPOINT commands;
  - ENTRYPOINT should be defined when using the container as an executable;
  - CMD should be used as a way of defining default arguments for an ENTRYPOINT command or for executing an ad-hoc command in a container;
  - CMD will be overridden when running the container with alternative arguments;
- VOLUME creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers. The value can be a JSON array, *VOLUME ["/var/log/"]*, or a plain string with multiple arguments, such as *VOLUME /var/log* or *VOLUME /var/log /var/db* . The docker run command initializes the newly created volume with any data that exists at the specified location within the base image;
- USER <UID>[:<GID>] sets the user name (or UID) and optionally the user group (or GID) to use when running the image and for any RUN, CMD and ENTRYPOINT instructions that follow it in the Dockerfile;

- WORKDIR sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile. If the WORKDIR doesn't exist, it will be created even if it's not used in any subsequent Dockerfile instruction. The WORKDIR instruction can be used multiple times in a Dockerfile. If a relative path is provided, it will be relative to the path of the previous WORKDIR instruction.
- ONBUILD adds to the image a trigger instruction to be executed at a later time, when the image is used as the base for another build. The trigger will be executed in the context of the downstream build, as if it had been inserted immediately after the FROM instruction in the downstream Dockerfile;
- STOPSIGNAL sets the system call signal that will be sent to the container to exit. This signal can be a valid unsigned number that matches a position in the kernel's syscall table, for instance 9, or a signal name in the format SIGNAME, for instance SIGKILL;
- HEALTHCECK tells Docker how to test a container to check that it is still working. This can detect cases such as a web server that is stuck in an infinite loop and unable to handle new connections, even though the server process is still running.

3.2.4. Docker containers

A Docker container is a runnable instance of an image[11]. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

Once the Docker image has been created by using the *docker build*, you can run it through the *docker run* command and then the following will happen:

- Docker creates a new container, as though you had run a docker container create command manually;
- Docker allocates a read-write filesystem to the container, as its final layer. This allows a running container to create or modify files and directories in its local filesystem;
- Docker creates a network interface to connect the container to the default network, since you did not specify any networking options. This includes assigning an IP address to the container. By default, containers can connect to external networks using the host machine's network connection;

[11] The content of this section has been taken, with some adaptations, from: Docker official website - Docker overview, https://docs.docker.com/get-started/overview/, October 1st, 2020; Docker official website - Docker run reference, https://docs.docker.com/engine/reference/run/, October 2nd, 2020.

- Docker starts the container and executes the CMD/ENTRYPOINT specified command.

Once the inner application ends its execution, the container stops but is not removed. You can start it again or remove it.

The *docker run* command takes this form:

docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]

Focusing on the OPTIONS, the following will be used afterwards in this thesis to implement the desired architecture:

- *-d* (detached) starts a container in detached mode. By design, containers started in detached mode exit when the root process used to run the container exits, unless you also specify the *--rm* option. If you use *-d* with *--rm*, the container is removed when it exits or when the daemon exits, whichever happens first;
- *--name* <name> specifies a name for the container created. You can use it to identify the container, i.e. if you want to remove it;
- *-p <port>* publishes the specified port;
- *-e <env_var_name=env_var_value>*sets the specified environment variable inside the container;
- *-v [host-src:]container-dest[:<options>]* mounts the *host-src* host volume on the *container-dest* container path. The container-dest must always be an absolute path such as /src/docs. The host-src can either be an absolute path or a name value. If you supply an absolute path for the host-dir, Docker bind-mounts to the path you specify. If you supply a name, Docker creates a named volume by that name. Note that certain host volumes, for security reasons, cannot be mounted inside a container. To force their mounting, the *--privileged* parameter must be specified. For example, the inodes related to the host devices cannot be mounted without the mentioned parameter. So, to mount the sound card inside the container you must execute the following command:

  docker run -v /dev/snd:/dev/snd --privileged [...]

- *--device* allows you to run devices inside the container without the --privileged flag. By default, Docker containers are "unprivileged" and cannot, for example, run a Docker daemon inside a Docker container. This is because by default a container is not allowed to access any devices, but a "privileged" container is given access to all devices. When the operator executes *docker run --privileged*, Docker will enable access to all devices on the host. So, to mount the sound card inside the container without using the *--privileged* parameter seen above, you can execute the following command:

  docker run --device=/dev/snd:/dev/snd [...]

3.3. Kubernetes

3.3.1. Introduction

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation[12]. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available. As mentioned in the previous sections, in the past there was an evolution in the execution of the server-side application. At first physical server has been used, then virtualization through the Virtual Machines usage was introduced, followed by the containerization as explained above.

Containers are a good way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start.

Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more.

Kubernetes provides you with:

- *Service* discovery and load balancing. Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable;
- Storage orchestration. Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more;
- Automated rollouts and rollbacks. You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate.
- Automatic bin packing. You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
- Self-healing. Kubernetes restarts containers that fail, replaces containers, kills containers that does not respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.
- Secret and configuration management. Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.

Kubernetes has several components and in the following sections we will examine those used to implement the desired architecture.

[12] The content of this section has been taken, with some adaptations, from: Kubernetes official website - What is Kubernetes, https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, October 2nd, 2020;

3.3.2. Pods

The *Pods* are the smallest deployable units of computing that you can create and manage in Kubernetes[13]. A Pod (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage/network resources, and a specification for how to run the containers. Each Pod is assigned a unique IP address for each address family. Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled. As well as application containers, a Pod can contain init containers that run during Pod startup. The shared context of a Pod is a set of Linux namespaces, cgroups, and potentially other facets of isolation - the same things that isolate a Docker container. Within a Pod's context, the individual applications may have further sub-isolations applied. In terms of Docker concepts, a Pod is similar to a group of Docker containers with shared namespaces and shared filesystem volumes. Usually it is not required to create Pods directly but through workloads resources such as *Deployment* or *Jobs*. In fact, Pods are designed as relatively ephemeral, disposable entities. When a Pod gets created (directly by you, or indirectly by a controller), the new Pod is scheduled to run on a Node in your cluster. The Pod remains on that node until the Pod finishes execution, the Pod object is deleted, the Pod is evicted for lack of resources, or the node fails. We said that a Pod can be created by hands or by a controller. In the latter case, you can use workload resources to create and manage multiple Pods for you. A controller for the resource handles replication and rollout and automatic healing in case of Pod failure. For example, if a Node fails, a controller notices that Pods on that Node have stopped working and creates a replacement Pod. The scheduler places the replacement Pod onto a healthy Node. An example of workload resource is the Deployment which will be provided in detail in the next section.

Controllers for workload resources create Pods from a pod template and manage those Pods on your behalf. PodTemplates are specifications for creating Pods, and are included in workload resources such as Deployments. Each controller for a workload resource uses the PodTemplate inside the workload object to make actual Pods. The PodTemplate is part of the desired state of whatever workload resource you used to run your app.

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: hello-world-job
  labels:
    app: hello-world
spec:
  template:
    #Pod template begin
    metadata:
      labels:
        app: hello-world
    spec:
      nodeName: worker-01
      containers:
        - name: hello-world
          image: busybox
```

[13] The content of this section has been taken, with some adaptations, from: Kubernetes official website - Pods, https://kubernetes.io/docs/concepts/workloads/pods/, October 7th, 2020; Kubernetes official website - Pod lifecycle, https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/, October 7th, 2020.

```
        env:
          - name: ENV_MESSAGE
            value: "Hello world!"
        volumeMounts:
          - mountPath: /tmp/hostname
            name: hostname
          - mountPath: /dev/snd
            name: sound
        securityContext:
          privileged: true
        command: ['sh', '-c']
        args:
          - 'echo "hostname=$(cat /tmp/hostname)" && echo "env_message=$ENV_MESSAGE"'
        ports:
          - name: ssh-cont-port
            containerPort: 22
    volumes:
      - name: hostname
        hostPath:
          path: /etc/hostname
      - name: sound
        hostPath:
          path: /dev/snd
    restartPolicy: OnFailure
    #Pod template end
```

The code snippet above is an example of a simple Job with a pod template that creates one container. The template begins with *metadata* specification which includes the *"app"* label definition. That label will be bound to the pod itself make it retrievable through the command below:

kubectl get pod -l app=hello-world

The template definition continues with the actual Pod definition which begins with the *nodeName* specification. This is used to tell Kubernetes the cluster node name in which the pod must be scheduled. If there is not a node having that name, the pod will not be scheduled at all. This is a possible way to force the Kubernetes behavior once it has to schedule pods.

Then, there is the *containers* specification which is a list of containers each one including the following fields:

- *name*, the name given to that container;
- *image,* the image that will be used to create the container itself;
- *env,* a list of *name-value* couples which represent each one a container environment variable;
- *volumeMounts*, a list of volumes that have to be mounted inside the container. In the simplest form, just the *mountPath* and the *name* need to be specified. The first one represents the container path in which the volume has to be mounted, the second one is bound to the *volumes* definition that will be shown afterwards;
- *securityContext,* allows specifying whether a privileged context need to be used to mount the volumes listed in *volumeMounts*. As described in section 3.2.4., it is required to use a privileged security context to properly mount certain host volumes, i.e. the inodes related to host devices such as the sound card one in the */dev/snd* path. Docker provides both the *-v [...] --privileged* and the *--device* parameters of the *docker run* command for this purpose. Unfortunately, Kubernetes does not have a *--device* correspondent thus the *privileged* security context might be used;

- *command*, specifies the command that needs to be executed inside the container once the pod starts. This field is not always required since the Docker image has its own CMD or ENTRYPOINT specification. If it will be specified, it will override the mentioned Docker command or entry point;
- *args*, defines a list of arguments to be passed to the command specified with the field above or to the Docker CMD/ENTRYPOINT defined command;
- *ports*, specifies a list of port that have to be exposed to make the pod reachable from any node in the cluster. To expose that ports outside the cluster, Kubernetes *services* should be used.

The Pod template definition continues with the shared *volumes* specification. Actually, the *volumes* field is a list of named volumes that can be mounted (through the *volumeMounts*) inside each Pod's container. There are several kinds of volumes, those shown in the example above are both *hostPath* which enable to mount the host's paths specified through the *path* fields inside the pod's container.

Finally, there is the *restartPolicy* field, useful to specify the Kubernetes behavior in case the pod fails its execution.

Note that in the case shown above the pod will have just one container, but it is possible to obtain a multi-container pod by specifying more containers inside the *containers* list. Moreover, there are several other pod's specifications that have not been discussed above since they will not be used in this thesis.

The pods follow a defined lifecycle, starting in the *Pending* phase, moving through *Running* if at least one of its primary containers starts OK, and then through either the *Succeeded* or *Failed* phases depending on whether any container in the Pod terminated in failure. Whilst a Pod is running, the kubelet is able to restart containers to handle some kind of faults. Within a Pod, Kubernetes tracks different container states: *Waiting, Running* and *Terminated*. Moreover, the Pod has a PodStatus as well which has an array of PodConditions through which the Pod has or has not passed:

- *PodScheduled*, the Pod has been scheduled to a node;
- *ContainersReady*, all containers in the Pod are ready;
- *Initialized*, all init containers have started successfully;
- *Ready*, the Pod is able to serve requests and should be added to the load balancing pools of all matching Services.

The conditions above are ideal to check whether the pod is able to serve requests before proceeding with the execution of something that needs that pod to work properly. This can be done by waiting for the *Ready* condition through the following command:

```
kubectl wait --for=condition=Ready pod -l “app=hello-world”
```

3.3.3. Deployments and ReplicaSets

A Deployment provides declarative updates for Pods ReplicaSets[14] whose purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods. A ReplicaSet is defined with fields, including a selector that specifies how to identify Pods it can acquire, a number of replicas indicating how many Pods it should be maintaining, and a pod template specifying the data of new Pods it should create to meet the number of replicas criteria. A ReplicaSet then fulfills its purpose by creating and deleting Pods as needed to reach the desired number. When a ReplicaSet needs to create new Pods, it uses its Pod template. Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to Pods along with a lot of other useful features. Therefore, it is recommended using Deployments instead of directly using ReplicaSets, unless it is required a custom update orchestration or updates are not required at all.

It is possible to create a Deployment as shown in the code snippet below.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 5
  selector:
    matchLabels:
      app: nginx
  template:
    #Pod template begin
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
    #Pod template end
```

In the example above a Deployment named *nginx-deployment* is created which in turn will create a ReplicaSet to bring up five *nginx* pods.

The *replicas* field specifies that the Deployment has to create five replicated pods.

The *selector* shown above defines how the Deployment finds which Pods to manage. In this case, you simply select a *label* that is defined in the Pod template (*app: nginx*). However, more sophisticated selection rules are possible, as long as the Pod template itself satisfies the rule.

The *template* defined above is actually the Pod template whose description has been discussed deeper in the previous section.

[14] The content of this section has been taken, with some adaptations, from: Kubernetes official website - Deployments, https://kubernetes.io/docs/concepts/workloads/controllers/deployment/, October 7th, 2020; Kubernetes official website - ReplicaSet, https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/, October 7th, 2020.

Note that the *template* and the *selector* are the only required fields. The *replicas* field is optional, and it defaults to one. The *selector* is required since it specifies a label selector for the Pods targeted by that Deployment.

Assuming that the Deployment definition above has been stored inside the *nginx-deployment.yaml* file, it can be created by running the following command:

```
kubectl apply -f nginx-deployment.yaml
```

A Deployment can be updated, i.e. by modifying the image version, through the following command:

```
kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1 --record
```

This way, all the pods related to that Deployment will be updated to the new image version, ensuring that only a certain number of Pods are down while they are being updated. By default, it ensures that at least 75% of the desired number of Pods are up (25% max unavailable). Moreover, it is possible to rollback a Deployment for example, when the Deployment is not stable, such as crash looping. By default, all of the Deployment's rollout history is kept in the system so that you can rollback anytime you want.

Also, it is possible to scale a Deployment by using the following command:

```
kubectl scale deployment.v1.apps/nginx-deployment --replicas=10
```

This way five more pod replicas will be created to facilitate more load.

3.3.4. Jobs

A Job creates one or more Pods and ensures that a specified number of them successfully terminate[15]. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (i.e. Job) is complete. Deleting a Job will clean up the Pods it created.

A simple case is to create one Job object in order to reliably run one Pod to completion. The Job object will start a new Pod if the first Pod fails or is deleted (for example due to a node hardware failure or a node reboot). You can also use a Job to run multiple Pods in parallel.

A Job definition example has been shown in section 3.3.2. talking about pods templates.

Unlike the Deployment, the *selector* field is optional and in almost all cases it should not be specified.

Even in this case, a container in a Pod may fail for a number of reasons, such as because the process in it exited with a non-zero exit code, or the container was killed for exceeding a memory limit, etc. If this happens, and the .spec.template.spec.restartPolicy = "OnFailure", then the Pod stays on the node, but the container is re-run. Therefore, your program needs to handle the case when it is restarted locally, or else specify .spec.template.spec.restartPolicy = "Never".

[15] The content of this section has been taken, with some adaptations, from: Kubernetes official website – Jobs, https://kubernetes.io/docs/concepts/workloads/controllers/job/, October 7th, 2020.

An entire Pod can also fail, for a number of reasons, such as when the pod is kicked off the node (node is upgraded, rebooted, deleted, etc.), or if a container of the Pod fails and the .spec.template.spec.restartPolicy = "Never". When a Pod fails, then the Job controller starts a new Pod. This means that your application needs to handle the case when it is restarted in a new pod. In particular, it needs to handle temporary files, locks, incomplete output and the like caused by previous runs.

When a Job completes, no more Pods are created, but the Pods are not deleted either. Keeping them around allows you to still view the logs of completed pods to check for errors, warnings, or other diagnostic output. The job object also remains after it is completed so that you can view its status. It is up to the user to delete old jobs after noting their status. Delete the job with kubectl (e.g. *kubectl delete jobs/pi or kubectl delete -f ./job.yaml*). When you delete the job using kubectl, all the pods it created are deleted too.

By default, a Job will run uninterrupted unless a Pod fails (restartPolicy=Never) or a Container exits in error (restartPolicy=OnFailure).

3.3.5. Services and DNS for Services and Pods

A service is an abstract way to expose an application running on a set of Pods. With Kubernetes it is not needed to modify your application to use an unfamiliar service discovery mechanism since it gives Pods their own IP addresses and a single DNS name for a set of Pods. Moreover, Kubernetes can load-balance across those Pods.

Kubernetes Pods are nonpermanent resources created and destroyed to match the state of the cluster. If a Deployment has been used to run an application, it can create and destroy Pods dynamically. Even if each Pod has its own IP address, the set of Pods running in one moment in time could be different from the set of Pods running that application a moment later. This leads to a problem: if some set of Pods (call them "backends") provides functionality to other Pods (call them "frontends") inside your cluster, how do the frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload[16]? A Kubernetes *service* solve this problem since it is an abstraction which defines a logical set of Pods and a policy by which to access them (sometimes this pattern is called a micro-service). The set of Pods targeted by a Service is usually determined by a *selector*.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - name: nginx-port
    protocol: TCP
    port: 80
    targetPort: 80
  type: ClusterIP
```

[16] The content of this section has been taken, with some adaptations, from: Kubernetes official website – Service, https://kubernetes.io/docs/concepts/services-networking/service/, October 7th, 2020; Kubernetes official website – DNS for Services and Pods, https://kubernetes.io/docs/concepts/services-networking/dns-pod-service, October 7th, 2020.

The code snippet above shows the definition of a *service* named *nginx-service*. Assuming that the Deployment defined in section 3.3.3. has been created, this *service* targets the port 80 on any Pod related to that Deployment, since each of that Pods has the app=nginx *label*.

Kubernetes assigns this Service an IP address (sometimes called the "cluster IP"), which is used by the Service proxies.

The controller for the Service selector continuously scans for Pods that match its selector. A Service can map any incoming port to a targetPort. By default and for convenience, the targetPort is set to the same value as the port field.

Port definitions in Pods have names, and you can reference these names in the targetPort attribute of a Service. This offers a lot of flexibility for deploying and evolving your Services. For example, you can change the port numbers that Pods expose in the next version of your backend software, without breaking clients.

The default protocol for Services is TCP; you can also use any other supported protocol.

As many Services need to expose more than one port, Kubernetes supports multiple port definitions on a Service object. Each port definition can have the same protocol, or a different one.

For some parts of the applications (for example, frontends) it might be needed to expose a Service onto an external IP address, that is outside of your cluster. Kubernetes *ServiceTypes* allow you to specify what kind of Service you want. *Type* values and their behaviors are:

- *ClusterIP*. Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default ServiceType.
- *NodePort*: Exposes the Service on each Node's IP at a static port (the NodePort). A ClusterIP Service, to which the NodePort Service routes, is automatically created. You'll be able to contact the NodePort Service, from outside the cluster, by requesting <NodeIP>:<NodePort>.
- *LoadBalancer*: Exposes the Service externally using a cloud provider's load balancer. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.
- *ExternalName*: Maps the Service to the contents of the *externalName* field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up.

If you set the type field to *NodePort*, the Kubernetes control plane allocates a port from a range specified by --service-node-port-range flag (default: 30000-32767). Each node proxies that port (the same port number on every Node) into your Service. If you want a specific port number, you can specify a value in the *nodePort* field. The control plane will either allocate you that port or report that the API transaction failed. This means that you need to take care of possible port collisions yourself. You also have to use a valid port number, one that is inside the range configured for NodePort use. The following code snippet shows a *NodePort* type *Service* example in which a specific port number has been declared through the *nodePort* field.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - name: nginx-port
    protocol: TCP
    port: 80
    targetPort: 80
    nodePort: 30007
  type: NodePort
```

Services are very useful for the pod reachability, but we still need to deal with the Service IP addresses to interact with the application. To overcome this problem, you can (and almost always should) set up a DNS service for your Kubernetes cluster using an add-on.

A cluster-aware DNS server, such as CoreDNS, watches the Kubernetes API for new Services and creates a set of DNS records for each one. If DNS has been enabled throughout your cluster then all Pods should automatically be able to resolve Services by their DNS name.

For example, if you have a Service called *my-service* in a Kubernetes namespace *my-ns*, the control plane and the DNS Service acting together create a DNS record for *my-service.my-ns*. Pods in the *my-ns* namespace should be able to find it by simply doing a name lookup for *my-service* (*my-service.my-ns* would also work).

Pods in other namespaces must qualify the name as *my-service.my-ns*. These names will resolve to the cluster IP assigned for the Service.

Kubernetes also supports DNS SRV (Service) records for named ports. If the my-*service.my-ns* Service has a port named *http* with the protocol set to TCP, you can do a DNS SRV query for *_http._tcp.my-service.my-ns* to discover the port number for http, as well as the IP address.

Note that the full DNS record for a Service has the form:

my-svc.my-namespace.svc.cluster-domain.example

and for a named port has the form:

_my-port-name._my-port-protocol.my-svc.my-namespace.svc.cluster-domain.example

Moreover, even each Pod has a DNS resolution which has the form:

pod-ip-address.my-namespace.pod.cluster-domain.example.

For example, if a pod in the default namespace has the IP address 172.17.0.3, and the domain name for your cluster is cluster.local, then the Pod has a DNS name:

172-17-0-3.default.pod.cluster.local

Any pods created by a Deployment or DaemonSet exposed by a Service have the following DNS resolution available:

pod-ip-address.deployment-name.my-namespace.svc.cluster-domain.example

3.3.6. Secrets

Kubernetes Secret is an object that let you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. Storing confidential information in a Secret is safer and more flexible than putting it verbatim in a Pod definition or in a container image[17]. Both users and the system itself can create secrets.

A possible use case involving Secret is storing user credentials required by Pods to access a database. For example, a database connection string consists of a username and password. You can store the username in a file *./username.txt* and the password in a file *./password.txt* on your local machine.

So, you can create the Secret by executing the following command:

```
kubectl create secret generic db-user-pass
  --from-file=./username.txt \
  --from-file=./password.txt
```

The default key name is the filename. You may optionally set the key name using --from-file=[key=]source.

Once a secret has been created through the command above, it can be used with a Pod in three ways:

- as files in a volume mounted on one or more of its containers. The code snippet below shows how a secret can be mounted inside a Pod.

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: db-user-pass
```

As you can see, a *volume* named *foo* of type *secret* has been declared. That volume is bound to the *secret* itself through its name, "*mysecret*", which is specified in the

[17] The content of this section has been taken, with some adaptations, from: Kubernetes official website – Secrets, https://kubernetes.io/docs/concepts/configuration/secret, October 7th, 2020; Kubernetes official website – Managing Secret using kubectl, https://kubernetes.io/docs/tasks/configmap-secret/managing-secret-using-kubectl, October 7th, 2020.

secretName field and is equal to name of the *secret* to be mounted. Then, that volume will be mounted in the */etc/foo* path inside the container thanks to the *volumeMounts* entry named *foo*. Note that the *secret* has to be mounted as a read-only volume and when a secret currently consumed in a volume is updated, projected keys are eventually updated as well. This way the *secret* content will be available in the *username.txt* and *password.txt* files under the */etc/foo/* path inside the container;

- as container environment variable. The code snippet below shows how a secret can be exposed in a container as an environment variable.

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: redis
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: db-user-pass
            key: username.txt
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: db-user-pass
            key: password.txt
  restartPolicy: Never
```

  As you can see, two environment variables, *SECRET_USERNAME* and *SECRET_PASSWORD*, have been defined. They have been bound with the Secret keys through the *secretKeyRef* field in which the Secret name and the key name have been specified;

- by the kubelet when pulling images for the Pod. The *imagePullSecrets* field is a list of references to secrets in the same namespace. You can use an *imagePullSecrets* to pass a secret that contains a Docker (or other) image registry password to the kubelet. The kubelet uses this information to pull a private image on behalf of your Pod.

Finally, note that more Pods can share the same Secret.

3.3.7. Persistent Volumes and Persistent Volume Claims

3.3.7.1. Introduction

So far it has been discussed how Kubernetes manages the part related to the pods, as well as their creation through the other core objects and their execution. Nevertheless, this is not the only required feature as often it is necessary to store some data obtained from the application execution. The *PersistentVolume* subsystem provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed. To do this, it has been introduced two new API resources: *PersistentVolume* and *PersistentVolumeClaim*[18].

A *PersistentVolume* (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using *Storage Classes*. It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like *Volumes* but have a lifecycle independent of any individual Pod that uses the PV. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

A *PersistentVolumeClaim* (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., they can be mounted *ReadWriteOnce*, *ReadOnlyMany* or *ReadWriteMany*).

3.3.7.2. Persistent Volumes and Persistent Volume Claims lifecycle

PVs are resources in the cluster. PVCs are requests for those resources and also act as claim checks to the resource. The interaction between PVs and PVCs follows this lifecycle[19]:

1. Provisioning.
   There are two ways PVs may be provisioned: statically or dynamically.
   - In the static way, a cluster administrator creates a number of PVs. They carry the details of the real storage, which is available for the use by cluster users. They exist in the Kubernetes API and are available for consumption.
   - The dynamic way is used when none of the static PVs the administrator created match a user's *PersistentVolumeClaim* so, the cluster may try to dynamically provision a volume specially for the PVC. This provisioning is based on *StorageClasses*: the PVC requests a storage class and the administrator must have created and configured that class for dynamic provisioning to occur. Claims that request the class "" effectively disable dynamic provisioning for themselves.
2. Binding.
   A user creates, or in the case of dynamic provisioning, has already created, a *PersistentVolumeClaim* with a specific amount of storage requested and with certain access modes. A control loop in the master watches for new PVCs, finds a matching PV (if possible), and binds them together. If a PV was dynamically provisioned for a new PVC, the loop will always bind that PV to the PVC. Otherwise, the user will always get at least what it was asked for, but the volume may be in excess of what was requested. Once bound, *PersistentVolumeClaim* binds are exclusive, regardless of how they were bound. A PVC to PV binding is a one-to-one mapping, using a *ClaimRef* which is a bi-directional binding between the *PersistentVolume* and the *PersistentVolumeClaim*.
   Claims will remain unbound indefinitely if a matching volume does not exist. Claims will be bound as matching volumes become available. For example, a

[18] The content of this section has been taken, with some adaptations, from: Kubernetes official website – Persistent Volumes, https://kubernetes.io/docs/concepts/storage/persistent-volumes, October 9th, 2020.

[19] The content of this section has been taken, with some adaptations, from: Kubernetes official website – Persistent Volumes, https://kubernetes.io/docs/concepts/storage/persistent-volumes, October 9th, 2020.

cluster provisioned with many 50Gi PVs would not match a PVC requesting 100Gi. The PVC can be bound when a 100Gi PV is added to the cluster.

3. Using.
   Pods use claims as volumes. The cluster inspects the claim to find the bound volume and mounts that volume for a Pod. For volumes that support multiple access modes, the user specifies which mode is desired when using their claim as a volume in a Pod. Once a user has a claim and that claim is bound, the bound PV belongs to the user for as long as they need it. Users schedule Pods and access their claimed PVs by including a *persistentVolumeClaim* section in a Pod's volumes block.
4. Reclaiming.
   When a user is done with their volume, they can delete the PVC objects from the API that allows reclamation of the resource. The reclaim policy for a *PersistentVolume* tells the cluster what to do with the volume after it has been released of its claim. Currently, volumes can either be *Retained*, *Deleted*, or *Recycled.*
   - The *Retain* reclaim policy allows manual reclamation of the resource. When the *PersistentVolumeClaim* is deleted, the *PersistentVolume* still exists and the volume is considered "released". But it is not available yet for another claim because the previous claimant's data remains on the volume. An administrator can manually reclaim the volume with the following steps: deleting the *PersistentVolume* (the associated storage asset in external infrastructure, such as an AWS EBS, GCE PD, Azure Disk, or Cinder volume, still exists after the PV is deleted); manually cleaning up the data on the associated storage asset accordingly; manually deleting the associated storage asset, or if you want to reuse the same storage asset, creating a new *PersistentVolume* with the storage asset definition.
   - For volume plugins that support the *Delete* reclaim policy, deletion removes both the *PersistentVolume* object from Kubernetes, as well as the associated storage asset in the external infrastructure, such as an AWS EBS, GCE PD, Azure Disk, or Cinder volume. Volumes that were dynamically provisioned inherit the reclaim policy of their *StorageClass*, which defaults to *Delete*. The administrator should configure the *StorageClass* according to users'expectations; otherwise, the PV must be edited or patched after it is created.
   - If supported by the underlying volume plugin, the *Recycle* reclaim policy performs a basic scrub (rm -rf /thevolume/*) on the volume and makes it available again for a new claim. Note that the *Recycle* reclaim policy is deprecated. Instead, the recommended approach is using dynamic provisioning.

3.3.7.3. Creating Persistent Volume

The simplest *PersistentVolume* that can be created is the *hostpath*. Kubernetes supports hostPath for development and testing on a single-node cluster[20]. A *hostPath PersistentVolume* uses a file or directory on the Node to emulate network-attached storage.

In a production cluster, it is not advisable to use *hostPath*. Instead a cluster administrator would provision a network resource like a Google Compute Engine persistent disk, an NFS share, or an Amazon Elastic Block Store volume.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

The configuration file above specifies that the volume is at */mnt/data* on the cluster's Node. The configuration also specifies a size of 10 gibibytes and an access mode of *ReadWriteOnce*, which means the volume can be mounted as read-write by a single Node. It defines the *StorageClass* name *manual* for the PersistentVolume, which will be used to bind *PersistentVolumeClaim* requests to this *PersistentVolume*.

3.3.7.4. Creating Persistent Volume Claims

The next step is to create a *PersistentVolumeClaim*. Pods use *PersistentVolumeClaims* to request physical storage[21].

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

[20] The content of this section has been taken, with some adaptations, from: Kubernetes official website – Configure a Pod to Use a PersistentVolume for Storage, https://kubernetes.io/docs/tasks/configure-pod-container/configure-persistent-volume-storage, October 9th, 2020.

[21] The content of this section has been taken, with some adaptations, from: Kubernetes official website – Configure a Pod to Use a PersistentVolume for Storage, https://kubernetes.io/docs/tasks/configure-pod-container/configure-persistent-volume-storage, October 9th, 2020.

In the configuration file above, it has been declared a *PersistentVolumeClaim* that requests a volume of at least three gibibytes that can provide read-write access for at least one Node. After the *PersistentVolumeClaim* has been created, the Kubernetes control plane looks for a *PersistentVolume* that satisfies the claim's requirements. If the control plane finds a suitable *PersistentVolume* with the same *StorageClass*, it binds the claim to the volume.

3.3.7.5. Consuming Persistent Volume Claims in a Pod

The next step is to create a Pod that uses the a *PersistentVolumeClaim* as a volume[22].

```
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
        claimName: task-pv-claim
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
```

Notice that the Pod's configuration file specifies a *PersistentVolumeClaim*, but it does not specify a *PersistentVolume*. From the Pod's point of view, the claim is a volume.

3.3.8. Assigning Pods to Nodes

Once a pod has to be scheduled, it is possible to constrain a Pod to only be able to run on particular Node(s), or to prefer to run on particular nodes[23]. There are several ways to do this, and the recommended approaches all use label selectors to make the selection. Generally such constraints are unnecessary, as the scheduler will automatically do a reasonable placement (e.g. spread your pods across nodes, not place the pod on a node with insufficient free resources, etc.) but there are some circumstances where you may want more control on a node where a pod lands, for example to ensure that a pod ends up on a machine with an SSD attached to it, or to co-locate pods from two different services that communicate a lot into the same availability zone.

[22] The content of this section has been taken, with some adaptations, from: Kubernetes official website – Configure a Pod to Use a PersistentVolume for Storage, https://kubernetes.io/docs/tasks/configure-pod-container/configure-persistent-volume-storage, October 9th, 2020.

[23] The content of this section has been taken, with some adaptations, from: Kubernetes official website – Assigning Pods to Nodes, https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node, October 7th, 2020.

The simplest way is using the *nodeName* constraint, but it has some limitation. A more flexible way is using the Node Affinity. In the following sections both of those will be discussed in detail.

3.3.8.1. Using nodeName

nodeName is the simplest form of node selection constraint, but due to its limitations it is typically not used[24]. nodeName is a field of PodSpec. If it is non-empty, the scheduler ignores the pod and the kubelet running on the named node tries to run the pod. Thus, if *nodeName* is provided in the PodSpec, it takes precedence over *nodeSelector*, *Node isolation/restriction* and *Affinity and anti-affinity*.

Some limitations bound to the use of *nodeName* are:

- If the named node does not exist, the pod will not be run, and in some cases may be automatically deleted.
- If the named node does not have the resources to accommodate the pod, the pod will fail and its reason will indicate why, for example OutOfmemory or OutOfcpu.
- Node names in cloud environments are not always predictable or stable.

The following code snippet shows an example of *nodeName* field usage.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
  nodeName: kube-01
```

This way the pod will be scheduled only in a node named *kube-01.*

3.3.8.2. Using Node Affinity

Node affinity is conceptually similar to another form of node selector constraint, the *nodeSelector*. This one specifies a map of key-value pairs[25]. For the pod to be eligible to run on a node, the node must have each of the indicated key-value pairs as labels (it can have additional labels as well). The most common usage is one key-value pair.

The affinity/anti-affinity feature, greatly expands the types of constraints you can express. The key enhancements are:

- The affinity/anti-affinity language is more expressive. The language offers more matching rules besides exact matches created with a logical AND operation;

[24] The content of this section has been taken, with some adaptations, from: Kubernetes official website – Assigning Pods to Nodes, https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node, October 7th, 2020.

[25] The content of this section has been taken, with some adaptations, from: Kubernetes official website – Assigning Pods to Nodes, https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node, October 7th, 2020.

- you can indicate that the rule is "soft"/"preference" rather than a hard requirement, so if the scheduler can't satisfy it, the pod will still be scheduled;
- you can constrain against labels on other pods running on the node (or other topological domain), rather than against labels on the node itself, which allows rules about which pods can and cannot be co-located.

The affinity feature consists of two types of affinity, "node affinity" and "inter-pod affinity/anti-affinity". Node affinity is like the existing *nodeSelector* (but with the first two benefits listed above), while inter-pod affinity/anti-affinity constrains against pod labels rather than node labels, as described in the third item listed above, in addition to having the first and second properties listed above.

There are currently two types of node affinity, called:

1. *requiredDuringSchedulingIgnoredDuringExecution*
2. *preferredDuringSchedulingIgnoredDuringExecution*

The former specifies rules that must be met for a pod to be scheduled onto a node, while the latter specifies preferences that the scheduler will try to enforce but will not guarantee. The "*IgnoredDuringExecution*" part of the names means that if labels on a node change at runtime such that the affinity rules on a pod are no longer met, the pod will still continue to run on the node. Node affinity is specified as field *nodeAffinity* of field *affinity* in the PodSpec. The code snippet below shows how the Node Affinity can be used to force a pod to be scheduled on a node

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/e2e-az-name
            operator: In
            values:
            - e2e-az1
            - e2e-az2
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: another-node-label-key
            operator: In
            values:
            - another-node-label-value
  containers:
  - name: with-node-affinity
    image: k8s.gcr.io/pause:2.0
```

The node affinity rule above says the pod can only be placed on a node with a label whose key is *kubernetes.io/e2e-az-name* and whose value is either *e2e-az1* or *e2e-az2*. In addition, among nodes that meet that criteria, nodes with a label whose key is *another-node-label-key* and whose value is *another-node-label-value* should be preferred.

You can see the operator *In* being used in the example. The new node affinity syntax supports the following operators: *In*, *NotIn*, *Exists*, *DoesNotExist*, *Gt*, *Lt*. You can use *NotIn* and *DoesNotExist* to achieve node anti-affinity behavior.

Chapter 4 - KubernetesOnDesktop architecture

4.1. General architecture

As explained before, the aim The aim of this project is to develop an architecture in which it is possible to offload the execution of a desktop application to a remote cluster. The problem is that, unlike the applications that usually run on a remote server (regardless of whether we refer to a single server or a cluster), a desktop application usually has a Graphical User Interface and can produce sounds. For this reason, running the application in a remote machine is not sufficient because we want to interact with that from the local machine by using its own Graphical User Interface and by reproducing the application sounds without affecting the user experience.

To realize this kind of architecture, we decided to split the general scenario in two sides: the server-side and the client-side.

The server-side is the one in charge of executing the target application and the client-side is the one in charge of showing the application Graphical User Interface and reproducing the application sound. The picture below will depict how it should look like.

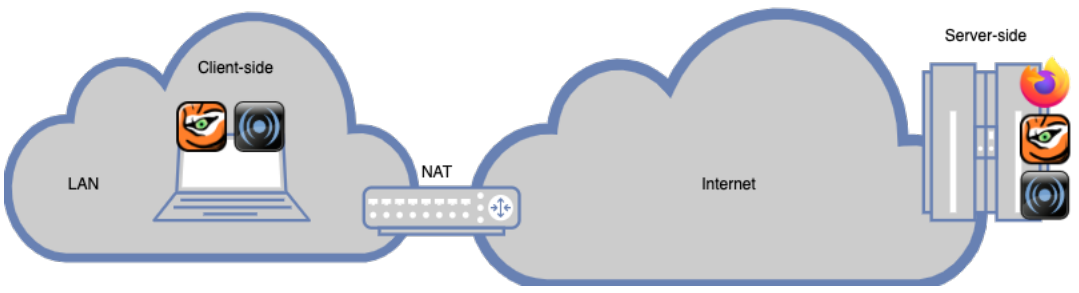

Figure 4: general architecture

Focusing on this picture, we can see that:

- the server-side consists in a block containing the desktop application, a VNC server (that will transmit the application Graphical User Interface video stream) and an audio client (which will forward the application sound);
- the client-side has a VNC client (that will receive the remote application Graphical User Interface video stream) and an audio server (that will receive the remote application audio stream).

4.2. Offloading the application execution

To realize the architecture above, the simplest solution may be to install natively in a remote machine a VNC server and all the target applications required by the user, and in the local machine a VNC viewer. Obviously, even if it is the simplest choice, this one has a lot of disadvantages:

1. it is required to install a huge amount of software in the remote machine (at least one for each target application, and the VNC server), and this might weigh the system and cause execution and configuration conflicts;

2. it is not possible to control how many remote machine resources will be assigned to each process, and this might cause crashes, unexpected behaviors and security issues;
3. this solution is not scalable, in fact to use a cluster instead of a single server, as we aim to, we need to schedule each process by hand. Also, we need to install in each machine all the required software, and this will cause undesired redundancy and a huge installation and maintenance overhead.

Much of those problems might be solved by just using Docker. In fact, as described in the previous chapters, Docker enables you to containerize your application and all its dependencies in a docker image. So, we can create a docker image that contains the target application and the VNC server, this way it is not required to install in each remote machine all the software but just the Docker Engine and it is possible to control the hosts accessible resources for each docker container.

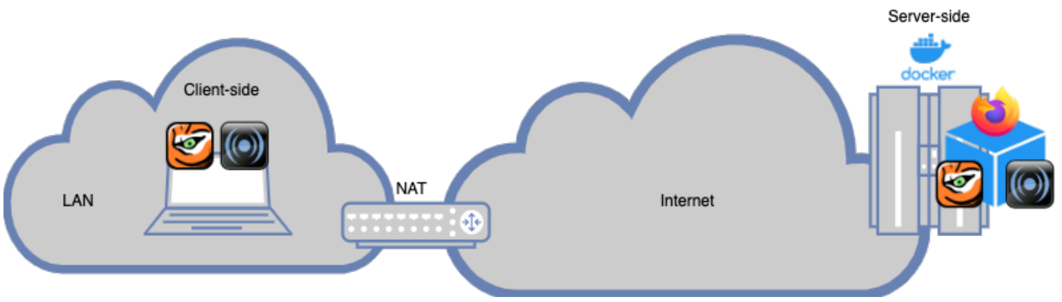

Figure 5: server-side containerization

The only unsolved problem is that if we want to schedule different processes in different remote machines, or even better in a cluster, we have to do it by hand, and this is not what we aim at. We can solve this just by using Kubernetes that, as described in the previous chapters, is a portable, extensible, open-source platform for managing containerized workloads and services. This way we can just create a pod, running the desktop application docker container, which will be automatically scheduled by Kubernetes in a machine inside the cluster.

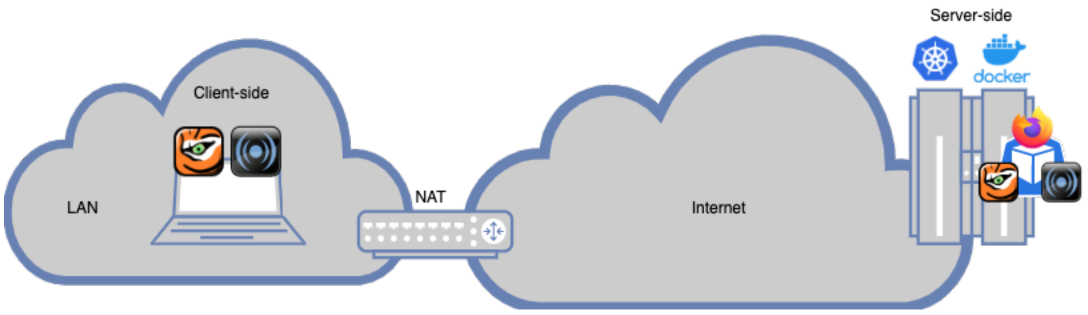

Figure 6: server-side cloudification

4.3. Containerizing the client-side

To complete the architecture described above we could think to fully containerize it by taking care of the client-side too. Actually, we can think of realizing the client-side with three different approaches:

1. installing natively the VNC viewer and the audio server;
2. creating a docker image containing the VNC viewer and the audio server, and run it on the local machine in its isolated environment;
3. running the VNC viewer and the audio server into a Kubernetes pod inside the cluster.

The first approach is the simplest one, even if we run the risk of falling in some of the problem seen for the server-side. In fact, including in this case, there may be some execution and configuration conflicts, and the loss of the assigned resource control which might cause crashes, unexpected behaviors and security issues. The Figure 6: server-side cloudification seen in the previous section depicts how it may appear.

The second approach, the one that realize a fully containerized architecture, is more difficult but resolves all the problems above. In this case, in fact, we'll have a docker image that contains both the VNC client and the audio server. This way the communication flows will happen between two containers, the one on the client-side and the one on the server-side (regardless of whether the latter is just a docker container or a Kubernetes pod). The picture below will depict how it might appear.

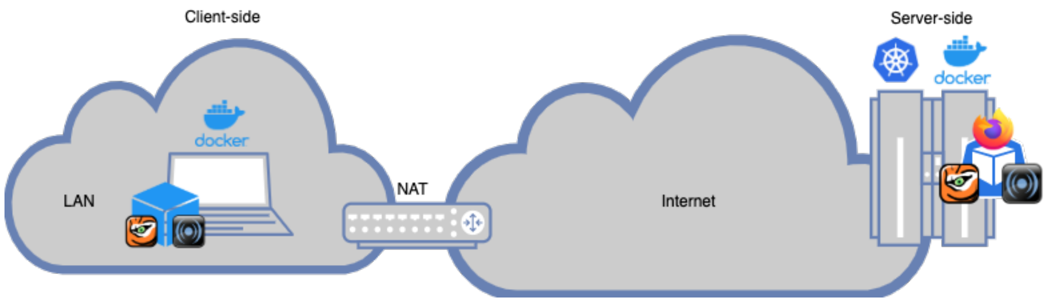

Figure 7: client-side containerization

The last approach, the one that realize a fully cloudified architecture, is a sort of evolution of the previous one because the client-side is already containerized but, in this case, the host that will receive the audio/video streams and will reproduce them is a Kubernetes cluster node itself. Obviously, this node must be equipped with a graphic card and a sound card in order to reproduce the audio/video streams and also with all the I/O peripherals (i.e. monitor, mouse, keyboard, speakers) in order to interact with the remote desktop application. The picture below will show how it might appear.

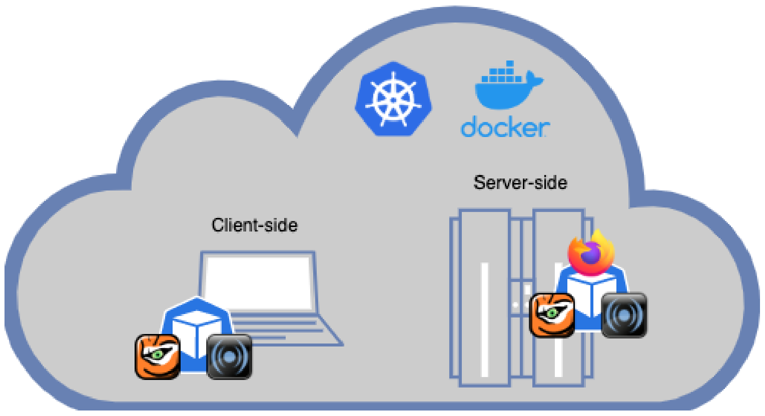

Figure 8: client-side cloudification

4.4. Fully cloudified architecture challenges

Since the fully cloudified architecture is completely based on Kubernetes pods, there are some challenges to face for its realization. In fact, to make it work properly, we need to take care of how the containers manage the audio/video streams (both in the server-side and in the client-side), and how the pod will be scheduled.

4.4.1. Managing the video streams

The first challenge is managing the video streams. To be more specific:

- in the server-side we need to use a windowing system to create the desktop application windows, and a VNC server that catches the video stream and redirects it to the VNC client;
- in the client-side we need to receive the video stream with a VNC viewer and to redirect it to the host graphic card.

Unlike the server-side, in which the host video interface is not involved since the container is headless and a containerized windowing system that doesn't require to be shown in that host will be used, in the client-side we need to show the Graphical User Interface and this means that we must interact someway with the host video interface.

Note that this challenge concerns not only the fully cloudified architecture but also the fully containerized architecture, that does not necessarily involve Kubernetes.

4.4.2. Managing the audio streams

A similar reasoning could be done for the audio streams. In fact:

- in the server-side we have a headless container which will not interact with the host sound interface. The only requirement is to forward the sound to the audio server inside the client-side;
- in the client-side, unlike the server-side, we need both to receive the sound from the remote desktop application through an audio server and to reproduce it in the host machine. So, in this case, we must interact someway with the host audio interface.

Note that, as for the video streams, this challenge concerns not only the fully cloudified architecture but also the fully containerized architecture.

4.4.3. Forcing the desktop application execution in a remote node

Since the pod scheduling is automatically carried out by Kubernetes and in the fully cloudified architecture the local machine is actually a Kubernetes cluster node, we run the risk of executing the desktop application pod inside the local machine. This condition, of course, should be avoided because it clashes with the aim of the project. In fact, what we want to realize is actually the application offload and it makes no sense to execute it in the local machine, even if containerized.

So, what we need to do is to find a way to force the desktop application pod scheduling and execution to another Kubernetes node.

4.4.4. Forcing the viewer execution in the local node

In the client-side we have similar situation, in fact, since the pod scheduling is automatically carried out by Kubernetes, we run the risk of executing the VNC viewer pod in a node that is not the local one. This is an undesirable event because even if we want to offload the desktop application execution, we still want to interact with it from the local machine.

So, this time, what we need to do is to find a way to force the VNC viewer pod scheduling and execution to the local Kubernetes node.

4.5. Requirements

The project implementing this architecture should meet some requirements in terms of start-up/execution times and of features to be supported as follows:

1. there should be the audio/video remotization;
2. the communications between components should be secured (for instance by using encryption) to avoid security issues;
3. the remote cluster response time should be less than 2 sec., which is at most 10 times the time required from an Operating System to schedule a process;
4. the remote desktop application pod should be up and running in less than 60 sec., which is at most 10 times the time required for a process to start;
5. the video stream should have a tunable image quality level;
6. the video stream should have a tunable compression level;
7. there should be configuration and data persistency for the remote application;
8. there should be scalability in terms of supported desktop application;
9. there should be the possibility to exploit some specific Kubernetes node hardware capabilities (i.e. use a NVIDIA graphic card) from the pod.

Chapter 5 - KubernetesOnDesktop: server-side implementation

5.1. Introduction

This section will talk about the server-side aspects. Then, in the next chapter, we will focus on the client-side implementation.

Before presenting an in-depth study of the implementation, an analysis of the protocols and tools chosen to realize the architecture above will be provided.

Note that this implementation, as well as the client-side one, is based on a previous proof of concept developed at Politecnico of Turin[26].

5.1.1. Tunneling and connection security protocols and tools

To make the connection secure we chose to use the SSH protocol and, in particular, we selected the software OpenSSH that «encrypts all traffic to eliminate eavesdropping, connection hijacking, and other attacks. In addition, OpenSSH provides a large suite of secure tunneling capabilities, several authentication methods, and sophisticated configuration options»[27] which are very useful for our purposes as we will see afterwards.

5.1.2. Video streaming protocols and tools

For the video streaming we choose to use the VNC protocol and, in particular, we selected two different implementations:

1. TigerVNC, which is a «high-performance, platform-neutral implementation of VNC (Virtual Network Computing), a client/server application that allows users to launch and interact with graphical applications on remote machines. TigerVNC provides the levels of performance necessary to run 3D and video applications, and it attempts to maintain a common look and feel and re-use components, where possible, across the various platforms that it supports»[28]. Moreover, TigherVNC provides a Xvnc server which is «based on a standard X server, but it has a "virtual" screen rather than a physical one. X applications display themselves on it as if it were a normal X display, but they can only be accessed via a VNC viewer»[29].
2. noVNC, that is «both a VNC client JavaScript library as well as an application built on top of that library. noVNC runs well in any modern browser including mobile browsers (iOS and Android)»[30]. This way it will be possible to interact with the remote application even from a browser.

[26] See: https://github.com/netgroup-polito/KubernetesOnDesktop.
[27] OpenSSH official website, https://www.openssh.com, September 9th, 2020.
[28] TigerVNC official website, https://tigervnc.org, September 9th, 2020.
[29] Tristan Richardson - RealVNC Ltd. and others, Xvnc – the X VNC server, https://tigervnc.org/doc/Xvnc.html, September 10th, 2020.
[30] noVNC official website, https://novnc.com/info.html, September 9th, 2020.

5.1.3. Audio streaming tools

For the audio streaming we choose to use PulseAudio which is a «sound system for POSIX OSes, meaning that it is a proxy for your sound applications. It allows you to do advanced operations on your sound data as it passes between your application and your hardware».[31] It is also able to do «things like transferring the audio to a different machine»[32]. This way it will be possible to catch the sound stream and forward it from the remote container to the local machine.

5.2. Docker implementation

5.2.1. Introduction

The server-side consists in two Dockerfiles that have been used to build, respectively, two docker images: the *base_image* and the *app_image*.

5.2.1.1. The base_image Dockerfile

The *base_image* Dockerfile installs inside the docker image the software required for the application remotization (but not the target application itself) and sets some environment variables.

To be more specific it will install the following software:

- Openbox, which is a free, lightweight and high configurable windows manager, that also requires the python-numpy library;
- x11-xkb-utils, which is a package containing graphical utilities, required to make the Graphical User Interface work properly;
- TigerVNC;
- noVNC, which also requires websockify that «just translates WebSockets traffic to normal socket traffic. Websockify accepts the WebSockets handshake, parses it, and then begins forwarding traffic between the client and the target in both directions»[33];
- PulseAudio;
- openssh-server, which is the OpenSSH server-side application.

Moreover, it will set the following environment variables related to the audio/video, the connection tunneling and connection security management:

- DISPLAY, which contains the display name. «From the user's perspective, every X server has a display name of the form: *hostname:displaynumber.screennumber*. This information is used by the application to determine how it should connect to the server and which screen it should use by default (on displays with multiple monitors). […] On POSIX systems, the default display name is stored in your DISPLAY environment variable»[34]. This variable will be used from the VNC server

[31] PulseAudio official website, https://www.freedesktop.org/wiki/Software/PulseAudio/, September 9th, 2020.
[32] Idem.
[33] Websockify official GitHub repository, https://github.com/novnc/websockify, September 9th, 2020.
[34] x(7) - Linux man page, https://linux.die.net/man/7/x, September 9th, 2020.

to specify which display it has to be linked to. In our case it has the *:0* value which is the default one;

- VNC_PORT, which is the TCP port the VNC server will listen to;
- NO_VNC_PORT, which is the TCP port the noVNC server will listen to;
- VNC_COL_DEPTH, which is the color depth of the visual to provide, in bits per pixel;
- DISPLAY_WIDTH, which is the desktop width;
- DISPLAY_HEIGHT, which is the desktop height;
- SSH_PORT, which is the TCP port the SSH server will listen to;
- VNC_PASSWORD, which is the VNC server password.

The following Dockerfile code snippet shows that variables declaration and their default values.

```
ENV DISPLAY=:0 \
    VNC_PORT=5900 \
    NO_VNC_PORT=5800 \
    SSH_PORT=22

[…]

ENV USER=vncuser \
    HOME=/home/vncuser \
    ROOT_WORKDIR=/opt/config \
    NO_VNC_HOME=/ \
    DEBIAN_FRONTEND=noninteractive \
    VNC_COL_DEPTH=24 \
    DISPLAY_WIDTH=1280 \
    DISPLAY_HEIGHT=1024 \
    VNC_PASSWORD=vncpassword \
    VNC_VIEW_ONLY=false \
    SECURE_CONNECTION=0
```

Note that those values could be overwritten, i.e. by the Kubernetes environment variable declaration.

5.2.1.2. The app_image Dockerfile

The *app_image* Dockerfile installs inside the docker image, created by using the *base_image* as parent image, the target desktop application and all its dependencies. Moreover, it installs the scripts which will be executed inside the container to run the server-side application, and all their related resources. To be more specific, the following files will be copied inside the container:

- *docker-entrypoint.sh* which, as its name suggests, is the docker entry point. It deals with the SSH service startup and launches the *user-startup.sh* script as a no privileged user;
- *user-startup.sh* which is the script that initializes and runs the VNC server, the openbox application and the target application;
- the openbox configuration files.

Since it is created by using the base_image as parent image, it will inherit from that all the previously installed software and the environment variables.

The *app_image* Dockerfile has been realized to be used as a sort of template to install the required target application inside the docker image. In fact, as you can see in the code snippet below, the *apt-get* package manager will be used to install the application specified in the APPLICATION build argument.

```
ARG APPLICATION=unknown
ARG REPO_TO_ADD=unknown
ENV APPLICATION=$APPLICATION

[...]

RUN if [[ ${REPO_TO_ADD} != "unknown" ]]; then \
        apt-get update && \
        apt-get install -y software-properties-common && \
        add-apt-repository ${REPO_TO_ADD}; \
    fi

RUN apt-get update
RUN apt-get install -y $APPLICATION
```

Moreover, since it could be possible that the target application package has its own repository and it is not available in the Linux Distribution official repositories, the REPO_TO_ADD build argument can be used to add the required repository. So, if no third-party repository is required, it is possible to build the *app_image* by running the command below (i.e. to install firefox):

```
docker build –build-arg APPLICATION=firefox -t firefox-headless-vnc
```

otherwise you can use the following command (i.e. to install blender):

```
docker build --build-arg APPLICATION=blender \
             --build-arg REPO_TO_ADD=”ppa:thomas-schiex/blender” \
             -t firefox-headless-vnc
```

Finally, the APPLICATION build argument will be exported as a container environment variable to let the *user-startup.sh* script know which target application to execute (you'll see that afterwards).

5.2.2. Managing the tunneling and the connection security

Once the docker container starts, the *docker-entripoint.sh* script will be executed. Its main purpose, as previously mentioned, is to start the SSH server. This server, actually, has two functions:

1. encrypting the connections between the server-side and the client-side;
2. creating tunnels between the server-side and the client-side.

As we can see afterwards, the first purpose is optional (i.e. to encrypt the VNC streams) and the second one is required in some cases (i.e. to forward the audio streams).

OpenSSH has some different authentication methods (i.e. password authentication, public key authentication, host-based authentication, ecc.). We chose to use the public key authentication one so that, on each execution, will be created a key pair (private and public keys) and then: the public one will be mounted someway inside the container running the target application; the private one will be used by the client-side to establish authenticated connections with the server-side.

```
chmod go-w $HOME
mkdir -p $HOME/.ssh
chmod 755 $HOME/.ssh
cp $HOME/ssh_secret/authorized_keys $HOME/.ssh
chmod 600 $HOME/.ssh/authorized_keys
chown $USER $HOME/.ssh/authorized_keys

service ssh start
```

As you can see in the *docker-entrypoint.sh* script code snippet above, first of all the SSH public key, which has been mounted inside the file *$HOME/ssh_secret/authorized_keys*, will be copied inside the directory *$HOME/.ssh*, that is the one in which the VNC server will look for the accredited public keys. Also, some files and folders owner and privileges must be changed to make the SSH server run properly. Then, the SSH server will be started, listening to the 22 TCP port (the SSH default one) for incoming connections.

5.2.3. Managing the video stream

As previously mentioned, the video stream managing has been realized through a combination of the Openbox windows manager, the VNC server and optionally the OpenSSH server. In the following sections we will see how they work and interoperate with each other.

5.2.3.1. Setting the VNC password

The first thing to do is to set a VNC password. This way, even though user chooses not to encrypt the connection, the VNC session, that in this case could be sniffed with a MITM attack, is still password protected. As you can see afterwards, the password is a One Time Token, generated before applying the Kubernetes `deploy` remotely, and exposed inside the container with the VNC_PASSWORD environment variable by overwriting the already declared one inside the *base_image* Dockerfile. This password will be used to connect to the container with both the protocols, VNC and noVNC.

```
mkdir -p "$HOME/.vnc"
PASSWD_PATH="$HOME/.vnc/passwd"

if [[ -f $PASSWD_PATH ]]; then
    rm -f $PASSWD_PATH
fi

[...]

echo "$VNC_PASSWORD" | vncpasswd -f >> $PASSWD_PATH
chmod 600 $PASSWD_PATH
```

As you can see in the *user-startup.sh* script code snippet above, at first all the default password values will be cleaned up to let the VNC server use the password specified with the new VNC_PASSWORD environment variable. Then, it will be encrypted with the *vncpasswd* command and then stored in the *$HOME/.vnc/passwd* file, which is the one from whom the VNC server will retrieve the password during its startup.

5.2.3.2. Cleanup the already running VNC server instances

Before starting the VNC server with the new password and with our own configuration parameters, we need to kill every VNC server instance that involve the display Unix socket we want to use. In fact, as described in section 5.2.1.1., the VNC server instance is always linked to a Unix socket specified inside the DISPLAY environment variable.

```
vncserver -kill $DISPLAY || rm -rfv /tmp/.X*-lock /tmp/.X11-unix
```

As you can see in the *user-startup.sh* script code snippet above, if already exists a VNC server instance linked to the display we want to use, it will be killed. If an error occurs (i.e. because some resources are locked or there is no VNC server running), all the temporary files related to the X service will be removed.

5.2.3.3. Starting the VNC servers

Before starting the VNC servers, the script checks if it is required to secure the connection through encryption. In this case, as we will see afterwards, from the client-side it will be created an SSH encrypted tunnel that maps a client-side port with the server-side one. This way the incoming connection acts as if it comes from the *localhost* itself and not from a remote host as actually it is. So, as you can see in the *user-startup.sh* script code snippet below, the *vncserver* command will be executed by specifing the *"-localhost"* parameter (through the IS_VNC_LOCALHOST variable) that forces it to accept *localhost* connection only.

```
if [[ $SECURE_CONNECTION -eq 1 ]]; then
   IS_VNC_LOCALHOST="-localhost"
fi
vncserver $DISPLAY $IS_VNC_LOCALHOST \
  -depth $VNC_COL_DEPTH -geometry ${DISPLAY_WIDTH}x${DISPLAY_HEIGHT} -noxstartup \
  -MaxDisconnectionTime=60

$NO_VNC_HOME/utils/launch.sh --vnc localhost:$VNC_PORT --listen $NO_VNC_PORT &
```

As you can see from the script above, both the VNC servers will be started as described below:

1. TigerVNC, through the command *vncserver* which is used to start both a Xvnc server, that virtualize a display reachable only through a VNC viewer, and then the VNC server itself. This command requires the following parameters:
   - *DISPLAY*, which specifies the Unix display socket the VNC server have to be linked to, in our case the virtual display which has "*:0*" value;
   - *-localhost*, added though the IS_VNC_LOCALHOST variable only if it is required to secure the connection, as described above;

- *-depth*, which specifies the color depth of the visual to provide, in bits per pixel;
- *-geometry,* which is a composition of display width and height and sets the image resolution;
- *-noxstartup*, which specifies not to automatically start a window manager in the TigerVNC session. This is required because, as we will see afterwards, we want to use Openbox as windows manager and we will run it by hands later;
- *-MaxDisconnectionTime*, which stops the server if no client has been connected (in our case, for 60 seconds);

2. noVNC, through its *launch.sh* script that requires the following parameters:
   - --vnc, which specifies the URL of an already running instance of a VNC server. In our case it will be specified the TigerVNC instance created before which is reachable through the *localhost:$VNC_PORT* URL;
   - *--listen*, which specifies the TCP port the noVNC server will listen to.

5.2.3.4. Running the windows manager and the target desktop application

Once the virtual display and the VNC servers are up and running, we are ready to execute the windows manager just by running the command in the following code snippet.

```
openbox-session &
```

This command first of all will read its configuration files then will run the windows manager. Actually, during the *app_image* building, the Openbox default configuration has been overwritten with an application-specific one (see the *app_image* Dockerfile code snippet below).

```
RUN rm -rf /etc/xdg/openbox && \
    cp -R openbox /etc/xdg/openbox && \
    mv /etc/xdg/openbox/${APPLICATION}.xml /etc/xdg/openbox/rc.xml
```

Once the Openbox windows manager is up and running, the target desktop application will be executed as you can see in the code snippet below.

```
if [[ ! -z "$APPLICATION" ]]; then
   exec $APPLICATION &
else
   pkill -P $$
fi
```

Note that the APPLICATION environment variable has been set in the *app_image* Dockerfile, according to the specified APPLICATION build argument, and that if the application doesn't exist (and this is an unexpected behavior since the application name should be the same of the installed one) each process will be killed and the script ends.

Once the target application is up and running, the script will wait for the first background process to terminate and then will kill all the remaining running processes, so it terminates its execution.

5.2.4. Managing the audio stream

As discussed above, the audio stream will be managed through the PulseAudio tool. It uses an environment variable named PULSE_SERVER that defines where the audio server is. It takes a protocol prefix like *unix:* or *tcp:* followed by the path or IP address of the server.

This is a very useful feature because we can set a remote server and use the TCP protocol. This way, we can install a PulseAudio server in the client-side and forward the sound, through a TCP connection, from the server-side to the client-side. So, the latter will be enabled to reproduce it on the local machine.

This could be a good solution but has two limitations:

1. we need to know in advance which the client-side IP address is;
2. the client-side machine should be reachable from the server-side. This is a problem, i.e. if the client-side machine is behind a NAT.

These limitations have been overcome by using an SSH tunnel. Actually, as we will see afterwards, the client-side will create an SSH tunnel and will map a remote TCP port (the one on the server-side) with a local TCP port (the one on the client-side). This way, each server-side connection to *localhost:ssh_server-side_port_mapped*, will be forwarded to the client-side. So, we can just set the PULSE_SERVER variable as *tcp:localhost:<port>* and the audio stream will be forwarded to the client-side.

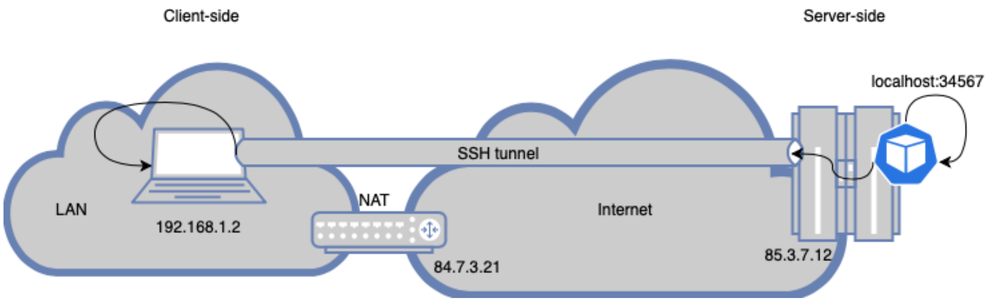

Figure 9: server-side to client-side PulseAudio port forwarding

The PULSE_SERVER environment variable, as we will see afterwards, will be set through the Kubernetes environment variable declaration.

5.2.5. Exploiting the host hardware capabilities

One of the main reasons of the application offloading is that the target application could require hardware resources which the user machine does not have. For example, to make a video rendering it should be desirable to use a powerful graphic card to reduce the execution time. So, a good feature to implement is to exploit the server-side hardware capabilities to obtain these results.

```
ARG FROM_IMAGE="ubuntu:18.04"
FROM ${FROM_IMAGE}
```

As you can see in the *base_image* Dockerfile code snippet above, it is possible to use a build argument named FROM_IMAGE to specify which *Parent Image* to use during the *base_image* building process. This way it is possible to use as *Parent Image* a docker image that contains specific drivers to exploit the remote hardware capabilities.

For example, if the remote node has a NVIDIA graphic card, it will be possible to use the NVIDIA provided *"nvidia/cuda:10.2-runtime-ubuntu18.04"* image as *Parent Image* instead of "*ubuntu:18.04*" which, as you can see in the code snippet above, is the default one.

Since we can obtain different *base_image*s depending on which *Parent Image* has been used during its building, also in the *app_image* Dockerfile there is a FROM_IMAGE build argument to specify which *base_image* to use as *app_image*'s *Parent Image*. See the *app_image* Dockerfile code snippet below.

```
ARG FROM_IMAGE="base-headless-vnc"
FROM ${FROM_IMAGE}
```

So, if you want to use a customized *base_image* you can build the *app_image* by executing the following command (i.e. to install blender and use a *base_image* that has the NVIDIA provided image as *Parent Image*):

```
docker build --build-arg APPLICATION=blender \
             --build-arg REPO_TO_ADD="ppa:thomas-schiex/blender" \
             --build-arg FROM_IMAGE=nvidia-base-headless-vnc \
             -t firefox-headless-vnc
```

5.3. Kubernetes implementation

5.3.1. Introduction

So far, we have analyzed the Docker server-side implementation. As we said in the previous chapter, that resolves almost all the issues we would have had if we had installed the target application natively on the server-side. The only unsolved problem is that if we want to schedule different processes in a cluster, we have to do it by hand. This has been solved by using Kubernetes.

The Kubernetes server-side implementation consists in:

1. a *deployment*, that will create a pod, which will run the *app_image* container described above, whose definition is contained in the *deployment.yaml* file;
2. a *service*, that makes the pod reachable both from the inside and the outside of the Kubernetes cluster, whose definition is contained in the *deployment.yaml* file too;
3. a *PersistentVolumeClaim*, to obtain the remote application configuration data persistency, whose definition is contained in the *volume.yaml* file.

Moreover, the application lifecycle (both for the server-side and the client-side) is managed through a bash script named *cloudify* which will create all the Kubernetes resources, will wait for their completion and then will clean up the cluster.

In the following sections we will see how the features above have been implemented by using Kubernetes and how have been managed through the *cloudify* script.

5.3.2. Managing the connection security

As described in the previous sections, we have to deal with two security levels:

1. the VNC password, which will be used even if there is not an encrypted connection between client and server;
2. the SSH key pair, which will be used if the user chooses to use an encrypted connection.

Talking about the first security level, as anticipated in section 5.2.3.1., the VNC password is a One Time Token generated from the *cloudify* script each execution.

```
function adjust_token {
  line=`grep -n 'VNC_PASSWORD' ${targetDeploy} | cut -d : -f -1`
  ((line=line+1))

  sed -i "${line}s/\".*\"/\"$token\"/" ${targetDeploy}
}

function start_deploy {

  [...]

  echo -n "Generating token..."
  token=$(cat /dev/urandom | tr -dc 'a-zA-Z0-9' | fold -w 32 | head -n 1)
  echo "OK"
  adjust_token

  [...]

}
```

As you can see in the *cloudify* code snippet above, the *start-deploy* function will generate a brand-new token by retrieving a 32 alpha-numeric chars string from the */dev/urandom* Unix inode, then it will be set inside the *deployment.yaml* file by using the *adjust_token* function.

```
apiVersion: apps/v1
kind: Deployment
[...]
spec:
  [...]
  template:
    [...]
    spec:
      [...]
      containers:
      - name: RAR_001_APP_NAME_PID-kubernetes
        image: RAR_002_APP_NAME-headless-vnc
        [...]
        env:
        [...]
        - name: "VNC_PASSWORD"
          value: "ECk01SFjMag8ENwDveZRwYrUnnLU1a3e"
```

The code snippet above shows that inside the *deployment* declaration there is the VNC_PASSWORD environment variable with the new generated token (set, as seen before, with the *cloudify* script *adjust_token* function) that will overwrite the one declared inside the container.

For the second security level, as anticipated in section 5.2.2., we chose to use the SSH protocol with the public key authentication. So, a key pair (private key and public key) will be created for each execution. Consequently, the public key will be mounted someway inside the container running the target application and the private one will be used by the client-side to establish authenticated connections with the server-side. Now we will explain how the key pair will be created and then how the public key will be mounted inside the container.

```
echo -n "Generating ssh key pair..."
ssh-keygen -t rsa -b 4096 -C "${USER}@kubernetes.io" \
           -f "${working_dir}/id_rsa" -N "" &>/dev/null
echo "OK"

[...]

echo -n "Creating ssh secret..."
kubectl create secret generic $ssh_secret_name \
        --from-file=authorized_keys="${working_dir}/id_rsa.pub" \
        -n $k8s_namespace &>/dev/null && ((state++))
```

As you can see in the *cloudify* code snippet above, first of all we have to create the SSH key pair by executing the *ssh-keygen* command and specifying the following parameters:

- -t rsa, which means that the RSA algorithm has to be used;
- -b 4096, which specifies the number of bits the key to create must have;
- -C, which just provides a comment;
- -f "…", which specifies the file name of the key file;
- -N "", which specifies an empty passphrase. This is required because we want to login to the remote container automatically by using the private key without any passphrase.

Then, a Kubernetes *secret* containing the public key will be created with the *kubectl create secret generic* command by specifying the following parameters:

- the secret name, which is stored inside the *$ssh_secret_name* variable and, as we will see afterwards, its name is parametrized to consent the applications concurrency management;
- --from-file, that specifies which is the file containing the public key created before;
- -n, which specifies which is the Kubernetes namespace we want to use.

This way we have a brand-new key pair and a secret containing the public key. Now the only thig left is let the pod running the target application to access it. This will be done, as you can see in the *deployment.yaml* code snippet below, by mounting the secret inside the container as a volume:

```
apiVersion: apps/v1
kind: Deployment
[...]
spec:
  [...]
```

```
template:
  [...]
  spec:
    [...]
    volumes:
    [...]
    - name: ssh-secret
      secret:
        secretName: RAR_003_SSH_SECRET
    [...]
    containers:
    - name: RAR_001_APP_NAME_PID-kubernetes
      image: RAR_002_APP_NAME-headless-vnc
      [...]
      env:
      [...]

      - name: "SECURE_CONNECTION"
        value: "1"
      [...]
      volumeMounts:
        [...]
        - name: ssh-secret
          mountPath: /home/vncuser/ssh_secret
          readOnly: true
```

This way, the public key will be mounted inside the container in the */home/vncuser/ssh_secret* path, which is the one from whom the *user-startup.sh* script inside the container will copy that key inside the local *$HOME/.ssh* folder. Moreover, to let the *user-startup.sh* script know whether the user chose to use an encrypted connection or not, as shown above, the *SECURE_CONNECTION* environment variable inside the container will be overwritten by a *deployment* environment variable declaration. This value will be set, according to the user choice, though the *adjust_encryption* function inside the *cloudify* script as shown in the code snippet below.

```
function adjust_encription {
        line=`grep -n 'SECURE_CONNECTION' ${targetDeploy} | cut -d : -f -1`
        ((line=line+1))

        sed -i "${line}s/[0-1]/${enc}/" ${targetDeploy}
}
```

Actually, there is also another last thing left: we need to expose the TCP 22 SSH port to make the pod reachable. This will be done by declaring it in the pod spec inside the *deployment.yaml* file. Anyway, since the ports declarations are managed all together by the same *cloudify* function used for the video stream port exposition, the *adjust_protocol* function, this will be shown in the following section.

5.3.3. Managing the video stream

The video stream managing is mainly done inside the docker container as we saw in section 5.2.3.. From the Kubernetes point of view, we just need to specify inside the deployment which port to expose depending on which protocol the user wants to use: VNC or noVNC. To be more specific, the *deployment.yaml* file, by acting as a template, contains all the ports to be exposed as shown in the code snippet below:

```
apiVersion: apps/v1
kind: Deployment
[...]
spec:
  [...]
  template:
    [...]
    spec:
      [...]
      containers:
      - name: RAR_001_APP_NAME_PID-kubernetes
        image: RAR_002_APP_NAME-headless-vnc
        [...]
        ports:
            - name: novnc-cont-port
              containerPort: 5800
            - name: vnc-cont-port
              containerPort: 5900
            - name: ssh-cont-port
              containerPort: 22
```

so, the *cloudify* script *adjust_protocol* function will remove the unused ones as shown below:

```
function adjust_protocol {
  from_novnc_service=`grep -w -n 'novnc-svc-port' ${targetDeploy} | cut -d : -f -1`
  ((till_novnc_service=from_novnc_service+3))
  from_vnc_service=`grep -w -n 'vnc-svc-port' ${targetDeploy} | cut -d : -f -1`
  ((till_vnc_service=from_vnc_service+3))
  from_novnc_container=`grep -w -n 'novnc-cont-port' ${targetDeploy} | cut -d : -f -1`
  ((till_novnc_container=from_novnc_container+1))
  from_vnc_container=`grep -w -n 'vnc-cont-port' ${targetDeploy} | cut -d : -f -1`
  ((till_vnc_container=from_vnc_container+1))

  if [[ $enc -eq 1 ]]; then
    sed -i "\
            ${from_novnc_container},${till_novnc_container}d;\
            ${from_novnc_service},${till_novnc_service}d;\
            ${from_vnc_container},${till_vnc_container}d;\
            ${from_vnc_service},${till_vnc_service}d" ${targetDeploy}
  elif [ "$protocol" = "vnc" ]; then
    sed -i "\
            ${from_novnc_container},${till_novnc_container}d;\
            ${from_novnc_service},${till_novnc_service}d" ${targetDeploy}
  else
    sed -i "\
            ${from_vnc_container},${till_vnc_container}d;\
            ${from_vnc_service},${till_vnc_service}d" ${targetDeploy}
  fi
}
```

To be more specific, as mentioned in the previous section, inside the *ports* declaration in the *deployment.yaml* file there is also the TCP 22 port used by the SSH protocol. So, depending on the user choice, the function will act as follows:

- if the encryption is enabled, all the ports will be removed except for the TCP 22 one;
- if the encryption is not enabled:
  - the TCP 22 port will be removed;
  - if the used protocol is VNC the noVNC port will be removed;
  - if the used protocol is noVNC the VNC port will be removed.

5.3.4. Managing the audio stream

The audio stream management is mainly done inside the docker container and from the architecture's client-side, as we have seen in section 5.2.4.. From the Kubernetes point of view, as mentioned in that section, we just need to specify the PULSE_SERVER environment variable to set right PulseAudio server URL, as shown in the following *deployment.yaml* code snippet.

```
apiVersion: apps/v1
kind: Deployment
[...]
spec:
  [...]
  template:
    [...]
    spec:
      [...]
      containers:
      - name: RAR_001_APP_NAME_PID-kubernetes
        image: RAR_002_APP_NAME-headless-vnc
        [...]
        env:
        - name: "PULSE_SERVER"
            value: "tcp:localhost:34567"
```

5.3.5. Managing the pod scheduling

Since the client-side machine, as it is for the server-side, could be a Kubernetes cluster node too, we need to force the server-side pod scheduling to a foreign node. To achieve this goal, the Kubernetes node affinity has been used.

```
apiVersion: apps/v1
kind: Deployment
[...]
spec:
  replicas: 1
  [...]
  template:
    [...]
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                - key: RAR_004_NS_LABEL
                  operator: DoesNotExist
```

As you can see in the *deployment.yaml* code snippet above, through the combination of the *matchExpression*, the label *key* and the *DoesNotExists* operator inside the *nodeAffinity*, it has been specified that the pod mustn't be scheduled in a node that has a certain label. This label, as we will see afterwards, has been parametrized to consent the applications concurrency management.

Thus, if the machine is part of the Kubernetes cluster, the *cloudify* script will add this label to the local node, as shown in the following code snippet.

```
function retrieve_curr_k8s_node_info {
  [...]
  IFS='#' read -r -a current_node_info <<< $(kubectl get nodes \
          -l kubernetes.io/hostname=`cat /etc/hostname` \
          -o 'jsonpath={range .items[0].status.addresses[*]}{.type}={.address}#{end}')
  for ((i=0; i<${#current_node_info[@]}; i++)); do
     if [[ $(echo ${current_node_info[$i]} | cut -d '=' -f 1) == "Hostname" ]];then
        current_node_name=$(echo ${current_node_info[$i]} | cut -d '=' -f 2)
     fi
        [...]
  done
  [...]
}

function prepare_cluster {
  [...]
  if [[ $run_mode -eq 2 ]]; then
     retrieve_curr_k8s_node_info
     [...]
     kubectl label no $current_node_name "$viewer_ns_label=true" --overwrite &>/dev/null
     [...]
  fi
}
```

As you can see above, first of all the node name of the local machine will be retrieved by using the */etc/hostname* file content. Actually, this will work properly with the default Kubernetes configuration, if the cluster administrator uses a node name that does not match the machine hostname there could be some issues.

Once the current node name has been retrieved, the node will be labeled with the *kubectl label no* command. Obviously, the *deployment* key must match the node label key so that Kubernetes will schedule the pod properly.

Finally, since the pod will be created from a *deployment*, it has been specified that we want just one replica of it as shown in the *deployment.yaml* code snippet above.

5.3.6. Managing the pod reachability

Once the server-side pod is up and running, we need to make it reachable both from the inside and the outside of the cluster depending on whether the client-side machine, respectively, is part of the cluster or not. To achieve this, a Kubernetes service has been defined.

```
apiVersion: v1
kind: Service
metadata:
  name: RAR_001_APP_NAME_PID-service
  labels:
    app: RAR_001_APP_NAME_PID
spec:
  selector:
    app: RAR_001_APP_NAME_PID
  ports:
  - name: novnc-svc-port
    protocol: TCP
    port: 5800
    targetPort: 5800
  - name: vnc-svc-port
    protocol: TCP
    port: 5900
    targetPort: 5900
  - name: ssh-svc-port
    protocol: TCP
    port: 22
    targetPort: 22
  type: ClusterIP
```

As you can see in the *deployment.yaml* code snippet above, a ClusterIP type service will be created and, since its definition acts as a template, as it was for the *deployment*, also in this case there is a function inside the *cloudify* script that will remove all the unused ports depending on the chosen protocol (VNC or noVNC) and the encryption enabling. Actually, the function is the same used for the deployment customization (*adjust_protocol*) so, depending on the user choices, also in this case, the function will act as follows:

- if the encryption is enabled, all the ports will be removed except for the TCP 22 one;
- if the encryption is not enabled:
  - the TCP 22 port will be removed;
  - if the used protocol is VNC the noVNC port will be removed;
  - if the used protocol is noVNC the VNC port will be removed.

The set of pods targeted by this service is determined by the *app* selector which matches the *pod* one. Note that, as we will see afterwards, the *app* selector has been parametrized, both in the *service* and in the *deployment* definition, to consent the applications concurrency management. Moreover, since the *deployment* replicas is set to one, even if the *pod* fails its execution, the restarted *pod* that will be reachable though the same service because it will have the same *app* selector of the previous one.

However, since a *ClusterIP* type *service* has been used so far, the *pod* will be reachable through the *service* only inside the cluster itself. To make it reachable from the outside the cluster, there is a function in the *cloudify* script that will replace the *ClusterIP* with a *NodePort* inside the *service* definition as shown in the following code snippet.

```
function enable_nodeport {
  sed -i "s/ClusterIP/NodePort/" ${targetDeploy}
}

function start_deploy {
  [...]
  if [[ "$run_mode" =~ ^[0-1]$ ]]; then
    enable_nodeport
  fi
  [...]
}
```

Note that, thanks to the *if-then-fi* statement above, only if the client-side machine is not part of the cluster the *NodePort* will be used.

5.3.7. Managing the data persistency

One of the project requirements is to have the remote application configuration and data persistency. To implement this feature, we choose to create a dynamically provisioned persistent volume claim which uses the *rook-ceph-block storageClassName.*

Before going into the implementation details, a brief introduction to this class will be provided. This is based on:

- *Rook,* which «is an open source cloud-native storage orchestrator, providing the platform, framework, and support for a diverse set of storage solutions to natively integrate with cloud-native environments»[35];
- *Ceph*, which «is a highly scalable distributed storage solution for block storage, object storage, and shared filesystems with years of production deployments»[36];
- *Block Storage*, which «allows a single pod to mount storage»[37].

The *rook-ceph-block* class is a combination of those three elements. Assuming that this class has been already defined inside the cluster, we can just create the *PersistentVolumeClaim* as shown in the code snippet below (i.e. for Firefox):

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: firefox-pv-claim
spec:
  storageClassName: rook-ceph-block
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
```

This definition will be reproduced for each target application to have a different volume for each one. In the case above a 100 Mibibyte volume, with read and write access privileges and named *firefox-pv-claim* will be created.

Once applied to the cluster, the *persistent volume claim* can be consumed by mounting it inside the pod as shown in the *deployment.yaml* code snippet below.

```
apiVersion: apps/v1
kind: Deployment
[...]
spec:
  [...]
  template:
    [...]
    spec:
      [...]
      volumes:
      [...]
#      - name: RAR_002_APP_NAME-persistent-storage
#        persistentVolumeClaim:
#          claimName: RAR_002_APP_NAME-pv-claim
      containers:
      - name: RAR_001_APP_NAME_PID-kubernetes
        image: RAR_002_APP_NAME-headless-vnc
        [...]
        volumeMounts:
          [...]
          #Firefox PVC claim
#          - name: firefox-persistent-storage
#            mountPath: /home/vncuser/.mozilla
          #Libreoffice PVC claim
#          - name: libreoffice-persistent-storage
#            mountPath: /home/vncuser/Documents
```

[35] Rook official website, https://rook.io/docs/rook/v1.4/, September 13th, 2020.
[36] Rook official website – Ceph Storage, https://rook.io/docs/rook/v1.4/ceph-storage.html, September 13th, 2020.
[37] Rook official website – Block Storage, https://rook.io/docs/rook/v1.4/ceph-block.html, September 13th, 2020.

As you can see above, both the *volumes* and the *volumeMounts* entries have been commented. This is because it is possible, both:

1. disabling the *persistent volume claim*, and nothing will be uncommented during the *cloudify* script execution;
2. enabling the *persistent volume claim* and, in this case:
   - the *volumes* entry will be uncommented and the *RAR_002_APP_NAME* string will be replaced with the target application name;
   - the *volumeMounts* entry related to the target application will be uncommented.

What explained above, will be done by the *cloudify* script as shown in the following code snippet:

```
function adjust_volumes {
   #Uncomment volumes
   from=`grep -n "RAR_002_APP_NAME-persistent-storage" ${targetDeploy} | cut -d : -f -1`
   ((to=from+2))
   sed -i "${from},${to} {s/^#//g}" ${targetDeploy}

   #Uncomment volumeMounts
   from=`grep -n "${application_name}-persistent-storage" ${targetDeploy} | cut -d : -f -1`
   ((to=from+1))
   sed -i "${from},${to} {s/^#//g}" ${targetDeploy}
}

function start_deploy {
    [...]
    if [ $use_pvc -eq 1 ]; then
      adjust_volumes
   fi
    [...]
}
```

5.3.8. Managing the applications concurrency

Since it is used a *deployment* to create the server-side *pod*, if we want to run more instances of the same application, we need to parametrize the resources name, as mentioned in the previous sections, to uniquely identify them. For this reason, for each resource name or label will be used a static part, shared with each application instance, and a parametrized part, which acts as a PID.

Thus, we need firstly to generate a PID and use it to compose the resources names as shown in the following *main* function code snippet below inside the *cloudify* script.

```
function main() {
         [...]
         app_pid=$(cat /dev/urandom | tr -dc 'a-z0-9' | fold -w 10 | head -n 1)
         ssh_secret_name="rar-ssh-secret-"$app_pid
         vnc_srv_app_pid="$application_name-$app_pid"
         [...]
         viewer_ns_label="kod/vncviewer-$app_pid"
    [...]
}
```

Similarly to the VNC_PASSWORD token generation seen in section 5.3.2., here the script will generate a brand-new PID by retrieving a 10 alpha-numeric chars string from the */dev/urandom* Unix inode. Then, by combining a static part with the brand-new PID as shown above, the following values will be created:

1. the secret name (*ssh_secret_name* variable), used to apply a new *secret* containing the SSH public key, as seen in section 5.3.2.. In this case also the *RAR_003_SSH_SECRET* string inside the secret *volumes* declaration of the *deployment* will be replaced;
2. the server-side application name + PID (*vnc_srv_app_pid* variable), used for:
   - the *deployment* name, the *deployment* "app" *selector* and *pod* and "app" *label* definitions, as shown in the *deployment.yaml* code snippet below;

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: RAR_001_APP_NAME_PID-deployment
spec:
  [...]
  selector:
    matchLabels:
      app: RAR_001_APP_NAME_PID
  template:
    metadata:
      labels:
        app: RAR_001_APP_NAME_PID
```

   - the *service* name, *service* "app" *label* and the *service* "app" *selector*, as shown in section 5.3.6. In this case the *RAR_001_APP_NAME_PID* string will be replaced;
3. the node *label* (*viewer_ns_label* variable), used for the *pod* scheduling seen in section 5.3.5.. In this case the *RAR_004_NS_LABEL* string will be replaced.

Then, the *cloudify* script will modify the *deployment.yaml* file by replacing the parameters with their values as described above by running the following function.

```
function adjust_appname_pid_secret_label {
   sed -i "s/RAR_001_APP_NAME_PID/${vnc_srv_app_pid}/" ${targetDeploy}
   sed -i "s/RAR_002_APP_NAME/${application_name}/" ${targetDeploy}
   sed -i "s/RAR_003_SSH_SECRET/${ssh_secret_name}/" ${targetDeploy}
   sed -i "s#RAR_004_NS_LABEL#${viewer_ns_label}#" ${targetDeploy}
}
```

This way, each *kubectl apply -f deployment.yaml* command will create a uniquely identified instance of each defined resources.

Chapter 6 - KubernetesOnDesktop: client-side implementation

6.1. Introduction

As we have seen in the KubernetesOnDesktop architecture chapter, the project consists in two parts: the server-side and the client-side. In the previous chapter we discussed the server-side by talking about its implementation focusing on the Docker and Kubernetes points of view and specifying, in each case, which issues have been solved and how.

In this chapter we will discuss the client-side implementation. The client-side might not have some of the server-side issues, i.e. to install a huge amount of software might not be required, as well as to control how many machine resources will be assigned to the process or to scale up the solution. Nevertheless, it may be useful to exploit some features of the containerized approach (i.e. to avoid configuration and execution conflicts) or of the cloudified approach (i.e. to exploit some of the Kubernetes features, if the viewer machine is a cluster node too), we decided to implement all the solutions to enable the user to choose which one fits his properly requirements. Note that, anyway, the native solution still remains subject to possible configuration and execution conflicts.

6.2. Native implementation

6.2.1. Introduction

The native implementation is completely managed by the *cloudify* script and requires the installation of both the *vncviewer* and the *vncserver*. To be more specific, it will be not necessary to install the entire *vncserver* package but one of its tools, the *vncpasswd*. Unfortunately, it has not a standalone installation. Moreover, we need to install an SSH client (in our case OpenSSH has been used) and a PulseAudio server which, often, are already installed in the most common linux distribution.

Finally, to install *kubectl* and to have a *config* file to access the remote cluster is required to let the *cloudify* script retrieve the remote *pod* information (i.e. the *node* IP, the *NodePort* exposed port, etc.), as we will see afterwards.

6.2.2. Managing the remote pod reachability

First of all, as we mentioned in the introduction above, we need to retrieve some remote *pod* and *service* information useful to interact with the server-side. To be more specific, the script requires the following parameters:

1. the IP address of the node executing the target application *pod*;
2. the name of the *pod* executing the target application;
3. the *namespace* to which that *pod* belongs;
4. the SSH port exposed through the *NodePort*;
5. if the connection encryption is not enabled, the VNC port exposed through the *Nodeport*.

Note that, as we will see afterwards:

- the SSH port exposed through the *NodePort* is always required since it will be used to create a tunnel for the audio forwarding;
- if the video encryption is enabled, the VNC port will not be retrieved because it will be not exposed at all by the server-side.

To retrieve the information above, it has been implemented a specific function in the *cloudify* script, as you can see in the code snippet above.

```
function retrieve_pod_info {
  read -r targetNodeIp targetPodName targetPodNamespace <<<$(kubectl \
   get pod -l app=$vnc_srv_app_pid -n $k8s_namespace \
   -o "jsonpath={..status.hostIP} {.items..metadata.name} {.items..metadata.namespace}")
  if [[ $enc -eq 1 ]]; then
   read -r targetNodePortSsh <<<$(kubectl \
     get svc -l app=$vnc_srv_app_pid -n $k8s_namespace \
     -o 'jsonpath={..spec.ports[?(@.name=="ssh-svc-port")].nodePort}')
  else
   read -r targetNodePortProtocol targetNodePortSsh <<<$(kubectl \
     get svc -l app=$vnc_srv_app_pid -n $k8s_namespace \
     -o 'jsonpath={..spec.ports[?(@.name=="'${protocol}'-svc-port")].nodePort}
{..spec.ports[?(@.name=="ssh-svc-port")].nodePort}')
  fi
}
```

This function will execute the *kubectl get* command by specifying the *jsonpath* output format through the *-o* parameter. Actually, JSONPath is a «template composed of JSONPath expressions enclosed by curly braces {}. Kubectl uses JSONPath expressions to filter on specific fields in the JSON object and format the output»[38].

6.2.3. Managing the video stream

Once all the required remote *pod* and *service* information are retrieved , we are ready to interact with the server-side.

As we have seen when we were talking about the server-side, the project implementation supports both VNC and noVNC protocols. Their implementation will be discussed separately afterwards but both share the same connection security infrastructure. So, first of all we will analyze it and then we will focus our attention on each specific video stream protocol.

6.2.3.1. Managing the video stream connection security

The video stream connection security is an optional feature which can be enabled by the user through a *cloudify* script execution parameter. In the client-side the connection security is also managed through the OpenSSH tool too but, obviously, this time the SSH client will be used.

Since, as said before, the connection security infrastructure is shared between both the VNC and the noVNC protocols and each of those protocol server is listening to different TCP port, we need to set which one to use depending on the chosen protocol by assigning to the *${port}* variable the right value. As you can see afterwards, this variable will be used by the *ssh*

[38] Kubernetes official website – JSONPath Support, https://kubernetes.io/docs/reference/kubectl/jsonpath/, September 15th, 2020.

command to implement the *ssh port forwarding*. The *cloudify* code snippet below will show you how *${port}* will be set.

```
function prepare_env_runmode_0_1 {
   [...]
   if [ $enc -eq 1 ]; then
      if [ "$protocol" = "vnc" ]; then
         port=5900;
      else
         port=5800;
      fi
   fi
   [...]
}
```

Once the *${port}* value has been set, we can establish the SSH connection by running the *ssh* command.

```
function run_native_vncviewver_novnc_application {
    [...]
    if [ $enc -eq 1 ]; then
        [...]
        ssh -o UserKnownHostsFile=/dev/null \
             -i "${working_dir}/id_rsa" \
             -o StrictHostKeyChecking=no -f -N \
             -S "${working_dir}/ssh_socket:${targetNodePortSsh}" \
             -L ${port}:localhost:${port} \
             vncuser@${targetNodeIp} \
             -p ${targetNodePortSsh} &>/dev/null
        [...]
   fi
    [...]
}
```

As you can see in the *cloudify* script code snippet above, when the connection security has been required (*$enc=1*) from the user*,* it will be obtained by executing the *ssh* command with the following parameters:

- *-o UserKnownHostsFile*, which specifies the host key checking file. «By default, the SSH client verifies the host key against a local file containing known, trustworthy machines. […] When you login to a remote host for the first time, the remote host's host key is most likely unknown to the SSH client. The default behavior is to ask the user to confirm the fingerprint of the host key»[39]. By setting it to */dev/null*, when a new host will be found, it will not be written in the default host key checking file;
- *-o StrictHostKeyChecking*, which «specifies if SSH will automatically add new host keys to the host key database file. By setting it to no, the host key is automatically added, without user confirmation, for all first-time connection. Because of the null key database file, all connection is viewed as the first-time for any SSH server host. Therefore, the host key is automatically added to the host key database with no user

[39] Peter Leung, How to disable SSH host key checking, https://linuxcommando.blogspot.com/2008/10/how-to-disable-ssh-host-key-checking.html, September 15th, 2020.

confirmation. Writing the key to the /dev/null file discards the key and reports success»[40];

- *-f,* which «requests ssh to go to background just before command execution. This is useful if ssh is going to ask for passwords or passphrases, but the user wants it in the background»[41] as we want to;
- *-N,* which specifies to «not execute a remote command. This is useful for just forwarding ports»[42];
- *-i,* which «selects a file from which the identity (private key) for RSA or DSA authentication is read»[43]. Since we choose to use the public key authentication method to login with SSH to the remote *pod,* here we will specify the private key related to the public one mounted inside the remote *pod*, as seen in section 5.2.3.2.;
- *-S,* which «specifies the location of a control socket for connection sharing»[44]. This has been used to link all together the SSH remote *pod* connection (to be more specific, the video and the audio connections), to clean them up easily at the end of the process execution;
- *-L port:host:hostport,* which «specifies that the given port on the local (client) host is to be forwarded to the given host and port on the remote side. This works by allocating a socket to listen to *port* on the local side […]. Whenever a connection is made to this port, the connection is forwarded over the secure channel, and a connection is made to *host* port *hostport* from the remote machine»[45]. This technique, named *port forwarding*, allows us to create a tunnel between the local host and the remote *pod* so that all the connections to the local *${port}* will be forwarded to the remote server, in our case, at the same port. Of course, as seen before, the *cloudify* script has already set the *${port}* value depending on whether the VNC or noVNC protocol will be used;
- *vncuser@${targetNodeIp}*, which specifies the user (*vncuser*) that wants to login the *pod* on the remote node having IP address *targetNodeIp*, which has been retrieved as seen in section 6.2.2.;
- *-p ${targetNodePortSsh}*, which specifies the port the SSH server is listening to. Note that its value has been retrieved as seen in section 6.2.2..

6.2.3.2. Managing the VNC protocol video stream

Since, as said before, the video stream could be whether secured by using an SSH tunnel or not, the target IP address and port will be different too. In fact:

- if a secured tunnel is used all the connection to the remote VNC server will be done by using the *ssh port forwarding*, as we have seen in section 6.2.3.1.. This means

[40] Peter Leung, How to disable SSH host key checking, https://linuxcommando.blogspot.com/2008/10/how-to-disable-ssh-host-key-checking.html, September 15th, 2020.
[41] ssh(1) - Linux man page, https://linux.die.net/man/1/ssh, September 15th, 2020.
[42] Idem.
[43] Idem.
[44] Idem.
[45] Idem.

that we can just use *localhost* as an IP address and as a port the one the server-side is listening to;

- if the connection security is disabled, the remote IP address will be the one of the Kubernetes node in which the server-side *pod* runs and the port is the one exposed through the *NodePort service*.

To obtain the right target IP address and port to establish the VNC connection, the *${target}* variable will be properly set and then used afterwards.

```
function prepare_env_runmode_0_1 {
  [...]
  if [ "$protocol" = "vnc" ]; then
     if [ $enc -eq 1 ]; then
        echo "Starting encrypted VNC connection..."
        target="localhost::${port}"
     else
        echo "Starting clear VNC connection..."
        target="$targetNodeIp::$targetNodePortProtocol"
     fi
  fi
}
```

As you can see in the *cloudify* script code snippet above:

- when the secure connection is enabled, the target will be set to *localhost::${port}* as mentioned before. The *${port}* variable is the one set in section 6.2.3.1.;
- when the secure connection in disabled, the target will be set to *$targetNodeIp::$targetNodePortProtocol*. The first variable will contain, as mentioned before, the IP address of the Kubernetes node which is running the server-side *pod*, the second one is the *NodePort service* exposed port. Note that these two variables have been already retrieved as seen in section 6.2.2..

Once the *$target* variable has been set, the *cloudify* script will just execute the following command to run the VNC viewer:

```
vncviewer -CompressLevel $compression -QualityLevel $quality $target \
-passwd <(echo ${token} | vncpasswd -f) 2>/dev/null
```

As you can see in the code snippet above, the following parameters have been used in the *vncviewer* command:

- -*CompressLevel* to specify the compression level;
- -*QualityLevel* to specify the image quality level;
- *$target*, the variable described above containing the remote IP address and the port in the form *IPaddress::port*;
- *-passwd* to specify the file containing the password required to access the VNC server. This password, as seen before, is a One Time Token generated as described in section 5.3.2. and stored in the *${token}* variable. Since this token is just a plaintext string and not an encrypted file as the *-passwd* would expect, it will be manipulated this way: the *vncpasswd*, by specifying the *-f* parameter, will receive the plaintext from the *stdin* through the pipe and will return the encrypted string to the *stdout* which will be returned as a file to the *-passwd* through the *"<"* operator.

By executing this command, a window showing the target application Grafical User Interface will appear and it can be used just as if the application was executed on the local machine.

The *vncviewer* is a blocking process, it means that the *cloudify* script will wait for the *vncviewer* execution end and then it will cleanup all the allocated resources.

6.2.3.3. Managing the noVNC protocol video stream

As seen for the VNC protocol, even in this case the video stream could be whether secured by using an SSH tunnel or not, so the target IP address and port will be different. For this reason, similarly to the previous case, we choose to use a variable named *$url* to obtain the right target IP address and port to establish the noVNC connection. However, this time, we does not need just a *IPaddress::port* string but a URL to be opened with a browser.

```
if [ $enc -eq 1 ]; then
    echo -n "Starting encrypted NOVnc connection..."
    url="http://localhost:$port"
else
    echo -n "Starting clear NOVnc connection..."
    url="http://$targetNodeIp:$targetNodePortProtocol"
fi
```

The *cloudify* script code snippet above shows how the *$url* will be set. Actually, the procedure is very similar to the one already seen in the previous section for the VNC protocol, except for the obtained string which has the form: *http://IPaddress:port* .

It means that we can do the same considerations about the *localhost:$port* (for the *ssh port forwarding*) and *$tagetNodeIp:$targetNodePortProtocol* usage and that the mentioned variables are exactly the same seen in the previous section for the VNC protocol.

Once the *$url* variable has been set, the *cloudify* script will run a browser which will open that URL.

```
echo "Your token is ${token}, insert it into your browser"
notify-send -t 10000 -a 'Kubernetes on Desktop' "One time Token" "$token"
firefox $url &>/dev/null
```

As you can see in the code snippet above, since also in this case to access the noVNC server the token is required, first of all it will be printed on the command line, with the *echo* command, and shown with a notification popup, through the *notify-send* command.

Then, a browser will be executed, in our case Mozilla Firefox, which will open the target URL, thanks to the *$url* parameter, and show a login page. So, we can just copy/paste the token and login to the noVNC server. Once the access will be granted, we can interact with the target application through the browser.

Note that *firefox* is a blocking process, it means that the *cloudify* script will wait for the *firefox* execution end and then it will cleanup all the allocated resources.

6.2.4. Managing the audio stream

As mentioned in section 5.2.4., to manage the audio stream, the client-side will run a PulseAudio server which will be reachable from the remote *pod* by using an SSH tunnel that will map a remote TCP port (the one on the server-side) with a local TCP port (the one on the client-side).

Since this is a native implementation of the client-side, we can assume that the PulseAudio is already running on the local machine, but we still need to configure it to be used as a TCP remote server. To achieve this goal, we just need to load the *module-native-protocol-tcp* PulseAudio module as shown in the *cloudify* code snippet below.

```
pactl load-module module-native-protocol-tcp port=${pulsePort} auth-ip-acl=127.0.0.1
```

This will make the PulseAudio local instance exploitable from a remote machine through a TCP connection. Below, the command above will be examined in-depth:

- *pactl*, an application that «can be used to issue control commands to the PulseAudio sound server»[46];
- *load-module*, a parameter telling *pactl* to «load the specified module with the specified arguments into the running sound server»[47];
- *module-native-protocol-tcp,* a module enabling the PulseAudio daemon to be reachable through the TCP protocol on the port specified with the *port* parameter;
- *port*, a parameter which specifies the port the PulseAudio server will listen to for remote connections. In our case, this will be set to *${pulsePort}* value which is the same used for the PULSE_SERVER environment variable definition seen in section 5.3.4.. Actually, this value (which is *34567*) has been hard coded inside the *cloudify* script and the *deployment.yaml* definition file;
- *auth-ip-acl=127.0.0.1*, a parameter used to authenticate the client through its IP address. The IP address authentication is usually a really bad idea because it is subject to many attacks (i.e. IP spoofing, Man In The Middle, etc.) but, since there is an SSH tunnel which wraps the connection, as we will see afterwards, in this case it will not be an issue to authorize connections from localhost. Moreover, the connections will only come from the localhost thanks to *reverse port forwarding* mechanism that we will see afterwards.

Once the PulseAudio daemon is ready to accept remote TCP connection, we need to create a tunnel to forward the audio stream from the project server-side to the client-side. In fact, as mentioned in section 5.2.4., since the PulseAudio server runs on the project client-side, this mechanism has some limits:

1. the project server-side (it means the remote *pod*) needs to know in advance which is the project client-side IP address;

[46] Ubuntu manpages, http://manpages.ubuntu.com/manpages/trusty/man1/pactl.1.html, September 16th, 2020.
[47] Idem

2. the project client-side machine should be reachable from the server-side (it means from the remote *pod*). This is a problem, i.e. if the project client-side machine is behind a NAT.

Through the tunnel creation we will overcome these limitations but, this time, we cannot create it exactly the same way we did, as seen in section 6.2.3.1., to secure the VNC connection. In fact, in that case the tunnel forwards a local port (it means a port of the project client-side) to the remote server; this time we want to forward a remote port to a local one in order that it is possible creating connections from the project server-side (the remote *pod*) to the project client-side. To obtain this, a mechanism named *ssh reverse port forwarding* will be used.

```
ssh -o UserKnownHostsFile=/dev/null \
    -i "${working_dir}/id_rsa" \
    -o StrictHostKeyChecking=no -f -N \
    -M -S "${working_dir}/ssh_socket:${targetNodePortSsh}" \
    -R ${pulsePort}:localhost:${pulsePort} \
    vncuser@${targetNodeIp} \
    -p ${targetNodePortSsh} &>/dev/null
```

The *cloudify* script code snippet above creates a tunnel with the *remote port forwarding* mechanism. As you can see, the ssh command parameters are almost the same we have seen in section 6.2.3.1. when we were talking about the *ssh port forwarding*, except for the *-R* parameter which replaces the *-L* one.

In fact, the *-R port:host:hostport* parameter «specifies that the given port on the remote (server) host is to be forwarded to the given host and port on the local side. This works by allocating a socket to listen to *port* on the remote side, and whenever a connection is made to this port, the connection is forwarded over the secure channel, and a connection is made to *host* port *hostport* from the local machine».[48] Moreover, as side effect, the audio streaming will be also encrypted through the SSH protocol.

Actually, there is another parameter which has not been used in video encryption: the -*M*. Actually, this parameter «places the ssh client into "master" mode for connection sharing»[49]. This way, each tunnel that shares the same *-S* parameter value will share the same connection that has been created from the master. This has two effects:

1. it is not required to create a new TCP connection for each tunnel, so the process needs less resources than having separate connections for each tunnel;
2. it is possible to close all the connection related tunnels gracefully through a *ssh* command.

The *-M* parameter appears in this *ssh* command and not in the video stream encryption one just because this code snippet will be run for first during the *cloudify* script execution, so this one will be the master and all the others will be the slaves.

[48] ssh(1) - Linux man page, https://linux.die.net/man/1/ssh, September 16th, 2020.
[49] Idem.

6.3. Docker implementation

6.3.1. Introduction

In the previous section the client-side native implementation has been examined. As said before, even if it is the simplest one, this might present some issues (i.e. configuration and execution conflicts). Thus, the user might choose a containerized solution. In this case we can have two choices: using docker or using Kubernetes. The latter solution, of course, will work only if the local machine is part of the Kubernetes cluster too and will be deepened afterwards. Moreover, that one is based, someway, on the docker implementation since it uses the same client-side docker image, making this solution a sort of evolution of the previous one. For now, we will focus on the docker implementation but, before going into the its details, it should be better to put in evidence some elements:

1. unlike the server-side, the client side has just one Dockerfile which will contain, as we will see afterwards, all the software required to run the Graphical User Interface, to reproduce the audio stream and to manage the connection security;
2. the docker image will contain a script named *run_vncviewer.sh* which is the container entry point;
3. since the script mentioned above actually reuses much of the *cloudify* script commands, it needs to receive some run parameters at startup time: many of these are always mandatory, the others are required only in certain cases that will be discussed afterwards. These parameters will be passed to the *run.vncviewer.sh* script as "<*variable_name*>=<*variable_value*>" couples, through the *docker run* command;
4. unlike the native implementation, just the VNC protocol has been used to implement this solution. In fact, the noVNC in the native implementation was effective because it avoided installing any software in the client-side host since it uses a browser instead of a natively installed VNC viewer. The containerized solution achieves this goal automatically so installing a browser inside the container to make the noVNC protocol available would be just an unnecessary overhead. Note that if both the noVNC protocol and the docker run mode *cloudify* script parameters will be specified, once the script will be launched it will return an error and will terminate its execution.

Compared to the native implementation, as you can imagine, in this solution we will have some more challenges since we need to share, someway, the host audio and video peripherals with the container to make the video stream visible and the audio stream audible to the user. We will discuss the relative solutions in detail in the following sections.

6.3.2. Managing the remote pod reachability

As seen in the native implementation, even in this case the first required thing is to retrieve some remote *pod* and *service* information, useful to make the docker container interact with the server-side. To be more specific, these are the required elements:

1. the IP address of the node executing the target application *pod*;
2. the name of the *pod* executing the target application;

3. the *namespace* which that *pod* belongs to;
4. the SSH port exposed through the *NodePort*;
5. if the connection encryption is not enabled, the VNC port exposed through the *Nodeport*.

Actually, the *cloudify* script itself is in charge of retrieving this information by executing the same function used in the native implementation, as seen in section 6.2.2.. What is new in this case, is that we need to pass the obtained information to the container, as mentioned in the previous section, through the *docker run* command. For convenience, we will see which one will be passed and how once we will specify how they will be exploited to implement the client-side features.

6.3.3. Managing the video stream

6.3.3.1. Managing the video stream connection security

Before going into the connection security implementation details, it is useful to point out some elements:

1. as for the native implementation, even in this case the OpenSSH client tool will be used to implement the connection security. It will be installed inside the container through the following Dockerfile command.

```
RUN DEBIAN_FRONTEND=noninteractive apt-get install -y \ [...]
    ssh \
    [...]
```

2. as seen in section 6.2.3.1., even in this case, the video stream connection security is an optional feature which can be enabled by the user through a *cloudify* script execution parameter;
3. as we will see afterwards, this feature implementation requires most of the steps discussed in the section mentioned before, but with some variations.

Now we can analyze the steps to go through to implement this feature. First of all, we need to let the *run_vncviewer.sh* script inside the container know whether the connection security has been enabled by the user or not.

```
function create_container_and_launch_vncviewer {
[...]
docker run [...]
   $tigervnc_docker `#docker image name` \
   enc=$1 `#enc` \
   [...]
[...]
}

create_container_and_launch_vncviewer \
   ${enc} ${compression} ${quality} ${target} \
   ${token} ${targetNodeIp} ${targetNodePortSsh} ${port}
```

The *cloudify* script code snippet above shows that, for this purpose, the *run-vncviewer.sh enc* parameter will be passed by setting it to the *cloudify* script *${enc}* variable value: if its value is equal to 1 the connection will be secured, otherwise, if it is equal to 0 a clear connection will be used. Note that the *enc* parameter is always mandatory.

The next step is to select the TCP port the VNC server is listening to. This is required to create the *ssh* tunnel by using the *ssh port forwarding* mechanism. In the native implementation both the VNC and noVNC protocols were available so the server port should be different depending on the chosen protocol. Since in this case just the VNC one is available, the port will be always the same that is the VNC protocol-related one. Anyway, to exploit the code reuse, exactly the same mechanism we have seen in the native implementation will be used. It means that the *cloudify* script *${port}* variable will be set and, for this purpose, the *cloudify* script itself will execute the *prepare_env_runmode_0_1* function exactly the same way we have seen in section 6.2.3.1..

Once the port has been selected, we need to pass this information to the *run_vncviewer.sh* script. This, as you can see in the *cloudify* script code snippet below, will be obtained by passing *enc_port* parameter, set to the *cloudify* script *${port}* variable value, to the *run-vncviewer.sh* script.

```
function create_container_and_launch_vncviewer {
[...]
docker run [...]
    $tigervnc_docker `#docker image name` \
    [...]
    enc_port=$8 `#port` \
    [...]
[...]
}

create_container_and_launch_vncviewer \
    ${enc} ${compression} ${quality} ${target} \
    ${token} ${targetNodeIp} ${targetNodePortSsh} ${port}
```

Note that the *enc_port* parameter is mandatory only if the connection security is enabled.

Once the script knows which is the port the VNC server is listening to, following what is described in section 6.2.3.1. for the native implementation, the next step should have been running the *ssh* command to create the tunnel with the *ssh port forwarding* mechanism. However, there are still some missing parameters inside the *run_vncviewer.sh* script which have to be passed through the *docker run* command as well.

```
function create_container_and_launch_vncviewer {
docker run -d --name $vnc_cli_app_pid \
    -v ${working_dir}:/home/vnc/ssh_id_rsa \
    $tigervnc_docker `#docker image name` \
    [...]
    target_node_ip=$6 `#targetNodeIp` \
    target_node_port_ssh=$7   `#targetNodePortSsh` \
    [...]
}

create_container_and_launch_vncviewer \
    ${enc} ${compression} ${quality} ${target} \
    ${token} ${targetNodeIp} ${targetNodePortSsh} ${port}
```

In fact, as you can see in the *cloudify* code snippet above, we need to let the *run_vncviewer.sh* script know:

1. the IP address of the remote node running the server-side *pod*. This information will be passed to the script through the *target_node_ip* parameter, set to the *cloudify* script *${targetNodeIp}* variable value, and it is mandatory only if the connection security is enabled;
2. the TCP port the remote SSH server is listening to. This information will be passed to the script through the *target_node_port_ssh* parameter, set to the *cloudify* script ${*targetNodePortSsh}* variable value, and it is mandatory only if the connection security is enabled;
3. the *ssh private key*. Actually, since the *ssh* command will look for a file containing the private key and the that key is stored inside a host machine file, we simply need to mount that file inside the container through the *docker run -v* parameter, as shown above.

Now that all the required parameters are available inside the container, we can execute the *ssh* command as follows.

```
ssh -4 -o UserKnownHostsFile=/dev/null \
    -i "${SSH_ID_RSA}/id_rsa" \
    -o StrictHostKeyChecking=no -f -N \
    -S "${SSH_SOCKET}/ssh_socket:${!target_node_port_ssh}" \
    -L ${!enc_port}:localhost:${!enc_port} \
    vncuser@${!target_node_ip} -p ${!target_node_port_ssh}
```

As you can see in the *run_vncviewer.sh* script code snippet above, the command is exactly the same we have analyzed in section 6.2.3.1.. Of course, the variables names are different since this one is a different script, but the meaning is the same.

Finally, you will notice that there is the *-4* parameter which is missing in the native implementation. This is required just to force *ssh* to use only the IPv4 protocol.

6.3.3.2. Managing the VNC video stream

Before going into the VNC video stream implementations details, even in this case, it is useful to point out some elements:

1. as for the native implementation, even in this case the TigerVNC viewer will be used. Actually, since the *vncpasswd* tool is required too, as seen in section 6.2.1., we need to install the entire TigerVNC package, not just the *vncviewer*;
2. to enable the *vncviewer* to show the Graphical User Interface it is required to install inside the container the *libgl1-mesa-dri* and the *libgl1-mesa-glx* libraries;
3. as for the connection security, even this feature implementation requires most of the steps discussed in the native implementation equivalent described in section 6.2.3.2., even if with some adaptations.

All the software described above will be installed inside the docker container as shown in the Dockerfile code snippet below.

```
RUN DEBIAN_FRONTEND=noninteractive apt-get install -y \
    [...]
    libgl1-mesa-dri \
    libgl1-mesa-glx

RUN wget -qO- https://dl.bintray.com/tigervnc/stable/tigervnc-1.8.0.x86_64.tar.gz | \
    tar xz --strip 1 -C /
```

Now we can analyze the steps to go through to implement this feature.

Since, as said before, the video stream could be whether secured by using an SSH tunnel or not, the target IP address and port will be different too. In fact:

- if a secured tunnel is used all the connection to the remote VNC server will be done by using the *ssh port forwarding*, as we have seen in the previous section. This means that we can just use *localhost* as IP address and as port the one the server-side is listening to;
- if the connection security is disabled, the remote IP address will be the one of the Kubernetes node in which the server-side *pod* runs and the port is the one exposed through the *NodePort service*.

So, the right target IP address and port, required to establish the VNC connection, will be obtained by executing the same *cloudify* script function *prepare_env_runmode_0_1* code snippet shown in section 6.2.3.2. for the native implementation. This way the *${target}* variable will be set, then we just need to pass this value to the script inside the docker container through the *target* parameter. Moreover, to execute the *vncviewer* command the following parameters are required too:

1. the video compression level, passed to the *run_vncviewer.sh* script through the *compression* parameter, set to the *cloudify* script *${compression}* variable value;
2. the image quality level, passed to the mentioned script through the *quality* parameter, set to the *cloudify* script *${quality}* variable value;
3. the VNC token used to allow the client to access the VNC server, passed to the mentioned script through the *token* parameter, set to the *cloudify* script *${token}* variable value.

```
function create_container_and_launch_vncviewer {
[...]
docker run -d --name $vnc_cli_app_pid \ [...]
          -v /tmp/.X11-unix/:/tmp/.X11-unix/ \
          [...]
          -v /dev/shm:/dev/shm \
          -v /var/run/dbus:/var/run/dbus \
          -e DISPLAY \
          $tigervnc_docker `#docker image name` \
          [...]
          compression=$2 `#compression` \
          quality=$3 `#quality` \
          target=$4 `#target` \
          token=$5 `#token` \
          [...]

[...]
}

create_container_and_launch_vncviewer \
   ${enc} ${compression} ${quality} ${target} \
  ${token} ${targetNodeIp} ${targetNodePortSsh} ${port}
```

The *cloudify* code snippet above shows how the mentioned parameters will be passed to the *run_vncviewer.sh* script. Moreover, as you can see above, to make the Graphical User Interface work properly, there are some more requirements:

1. the host's *shared memory* (*/dev/shm*) have to be mounted inside the container to avoid crashes;
2. the host's *D-Bus* (/dev/dbus) have to be mounted inside the container. The D-Bus is an inter-process communication mechanism required to avoid crashes too;
3. the host's */tmp/.X11-unix/* have to be mounted inside the container. This directory contains the video Unix-domain socket;
4. the *DISPLAY* environment variable has to be set. This variable select which video Unix socket should be used. Combined with the */tmp/.X11-unix/* allows the container to exploit the host's video interface.

Now that all the required parameters, volumes and environment variables are available inside the container, we can execute the *vncviewer* command as follows.

```
vncviewer -CompressLevel ${!compression} -QualityLevel ${!quality} \
              ${!target} -passwd <(echo ${!token} | vncpasswd -f)
```

As you can see in the *run_vncviewer.sh* script code snippet above, the command is exactly the same we have analyzed in section 6.2.3.2.. Obviously, the variables names are different since this one is a different script, but the meaning is the same.

6.3.4. Managing the audio stream

Before going into the audio stream management implementation details, even in this case, it is useful to point out some elements:

1. as for the native implementation, even in this case the PulseAudio will be used. This will be installed inside the container through the Dockerfile command below.

```
RUN DEBIAN_FRONTEND=noninteractive apt-get install -y \
    [...]
    pulseaudio \
    [...]
```

2. as for the connection security and the VNC video stream management, even this feature implementation requires most of the steps discussed in the native implementation equivalent described in section 6.2.4., even if with some adaptations.

Now we can analyze the steps to go through to implement this feature. Even in this case, as described in the section mentioned above, to manage the audio stream, the client-side will run a PulseAudio server which will be reachable from the remote *pod* by using an SSH tunnel that will map a remote TCP port (the one on the server-side) with a local TCP port (the one on the client-side).

Unlike the native implementation, this time we have not a PulseAudio daemon already running in the container. Actually, it should be possible to mount the host's Unix socket as we did for the video interface, but in this case, we would have some issues. In fact, since we need to load the TCP module to receive the remote *pod* audio stream through that protocol, we need to have an entire PulseAudio daemon instance running and this is not possible just by mounting the audio Unix socket. So, we choose to mount the host's sound device into the container and then run PulseAudio by using the actual host's audio interface directly instead of through the host's Unix socket. So, this is what is required to make the PulseAudio instance run properly:

- mounting the host's sound card device inside the container not just as a volume, as we did for the video socket, but as an actual device as shown in the following *cloudify* script code snippet;

```
docker run [...] \
         --device /dev/snd:/dev/snd \
         [...]
```

- copying some configuration files to enable the PulseAudio instance to discovery the mounted sound card.

```
COPY [ "./resources/pulseaudio_config/client.conf",
"./resources/pulseaudio_config/daemon.conf",
"./resources/pulseaudio_config/default.pa", "/etc/pulse/" ]
COPY [ "./resources/pulseaudio_config/avahi-daemon.conf", "/etc/avahi/" ]
RUN chmod 777 /etc/pulse/*
```

  As you can see in the Dockerfile code snippet above, the following files will be copied inside the container:

  a. *client.conf*, which contains some default PulseAudio configuration variable;
  b. *daemon.conf*, which contains some others default PulseAudio configuration variable;
  c. *default.pa*, which is the most important configuration file since it is the one that allows PulseAudio to load the module to interact with the mounted sound card. This will be done by adding in this file the following line:

```
load-module module-alsa-sink device=hw:0,0
```

  Once the files have been copied, the Dockerfile will change their privileges to make them work properly.

Since all the required elements have been already prepared from the Dockerfile as specified above, the *run_vncviewer.sh* script just needs to launch the PulseAudio daemon.

```
function launch_pulseaudio_server {
    [...]
    pulseaudio -D 1>&2
    [...]
}
```

The *run_vncviewer.sh* code snippet above will launch the PulseAudio daemon which will load its configuration files and will be ready to reproduce the sound. Note that the *-D* parameter is to tell the PulseAudio process to run as a daemon, detached from the terminal.

Now that we have a PulseAudio daemon running inside the container and exploiting the mounted host's sound card, we can proceed as it has been seen in section 6.2.4. talking about the native implementation. So, first of all, we need to load the *module-native-protocol-tcp* module to enable the PulseAudio daemon to be used as a TCP remote server as shown in the *run_vncviewer.sh* code snippet below.

```
function launch_pulseaudio_server {
    [...]
    pactl load-module module-native-protocol-tcp port=${PULSE_PORT} auth-ip-acl=127.0.0.1
    [...]
}
```

As you can see, the command is exactly the same we have analyzed in section 6.2.4.. Of course, the variables names are different since this one is a different script, but the meaning is the same. Note that the *${PULSE_PORT}* is actually an environment variable which has been set through the following Dockerfile code snippet.

```
ENV [...]\
    PULSE_PORT=34567
```

The *PULSE_PORT* variable has the same value set in the server-side *PULSE_SERVER* environment variable seen in section 5.3.4.. This is, of course, because that value represents the TCP port the PulseAudio daemon is listening to receive the remote audio stream.

Once the PulseAudio daemon is ready to accept remote TCP connection, it is necessary to create a tunnel to forward the audio stream from the project server-side to the client-side.

```
function launch_pulseaudio_server {
    [...]
    ssh -4 -o UserKnownHostsFile=/dev/null \
        -i "${SSH_ID_RSA}/id_rsa" \
        -o StrictHostKeyChecking=no -f -N \
        -M -S "${SSH_SOCKET}/ssh_socket:${!target_node_port_ssh}" \
        -R ${PULSE_PORT}:localhost:${PULSE_PORT} \
        vncuser@${!target_node_ip} -p ${!target_node_port_ssh} 1>&2
    [...]
}
```

As it can be seen in the *run_vncviewer.sh* code snippet above, even in this case the command is exactly the same we have analyzed in section 6.2.4.. Obviously, the variables names are different too since this one is a different script, but the meaning is the same.

This way the tunnel which will wrap the audio stream can been created, and everything is ready to reproduce the sound on the local machine.

6.3.5. Managing the applications concurrency

In the Kubernetes server-side implementation we have seen that since a *deployment* is used to create the server-side *pod*, if we want to run more instances of the same application, we need to parametrize the resources name to uniquely identify them. Actually, even in this case, we need to use a different execution container name to uniquely identify it and to gracefully clean up the allocated resources once the execution ends. For this purpose, we parametrized the container name as shown in the following *cloudify* script code snippet.

```
function create_container_and_launch_vncviewer {
  [...]
  docker run -d --name $vnc_cli_app_pid \
  [...]
}
```

As you can see in the code snippet above, the container name will be set through the *--name* run parameter. The *$vnc_cli_app_pid* variable is composed by a static part, which is a string hard coded inside the $*tigervnc_container_name* variable, and a PID which has been generated as shown in section 5.3.8. and stored in the *$app_pid* variable.

```
function main() {
   [...]
   app_pid=$(cat /dev/urandom | tr -dc 'a-z0-9' | fold -w 10 | head -n 1)
   [...]
   vnc_cli_app_pid="$tigervnc_container_name-$app_pid"
   [...]
}
```

As you can see in the *cloudify* code snippet above, the *$vnc_cli_app_pid* is just a concatenation of the two variables mentioned before interspersed with a dash character.

Note that the *$app_pid* variable has the same value of the server-side, this way we can simplify the *pod*s identification and associate the client-side to the server-side *pod*.

6.4. Kubernetes implementation

6.4.1. Introduction

In the previous sections the native and the containerized implementation have been examined. This chapter focuses on the last kind of implementation, the Kubernetes one, which allows us to realize a fully cloudified architecture. We will explain how it has been realized and how we took advantage of being a cluster node by exploiting some Kubernetes features.

This implementation consists in a Kubernetes *job*, defined in the *vncviewer.yaml* file, running the docker image discussed in the previous sections. Of course, this will work only if the local machine is a Kubernetes cluster node too, equipped with a graphic card and a monitor to show the target application Graphical User Interface, a sound card and a speaker to reproduce its sounds, a keyboard and a mouse to interact with it.

Finally, as mentioned for the server-side implementation in section 5.3.1., the application lifecycle (both for the server-side and the client-side) is managed through the *cloudify* bash script which will create all the Kubernetes resources, will wait for their completion and then will clean up the cluster.

6.4.2. Managing the remote pod reachability

Since the local machine is e Kubernetes node, unlike the native and docker implementation, we can exploit a Kubernetes feature to reach the remote *pod*.

In this case, as discussed in section 5.3.6., the server-side will not create a *NodePort service* to make the *pod* reachable but just a simple *ClusterIP*. For this reason, it is not required to retrieve the server-side *pod* and *service* information seen in section 6.2.2. since we will exploit the DNS for services and Pods Kubernetes feature.

In fact, each Kubernetes *services* should be reached through a URL formed as follows.

```
my-svc.my-namespace.svc.cluster-domain.example
```

So, in our case, to make the remote *pod* reachable, we just need to create that URL by replacing:

- *my-svc*, with the name of the service created in the server-side as we discussed in section 5.3.8.. This name has a variable part, stored in the *$vnc_srv_app_pid cloudify* script variable containing the target application name and a PID, and a static part, which is the string "*-service*". This way, the final name will appear in the following way: *firefox-01a23h56fj-service*. So, the *my-svc* string has to be replaced with a concatenation of the mentioned *cloudify* variable and the constant string;
- *my-namespace*, with the Kubernetes *namespace* the *service* belongs to. This value has been stored in the *cloudify* script variable named *$k8s_namespace*;
- *cluster-domain-example*, with the cluster domain. Actually, since we assume that the cluster has a single domain, the default one, we can remove this part of the URL.

```
function prepare_env_runmode_2 {
          [...]
          svc_URL=$vnc_srv_app_pid"-service."$k8s_namespace".svc"
          [...]
}
```

The *cloudify* script code snippet above shows how the mentioned URL has been composed and stored in the *$svc_URL* variable. So, each time we need the IP address of the remote node running the target application *pod* we can use the mentioned variable in place of it. We will see how it will be exploited and in which case afterwards, once we will describe how have been implemented the client-side features.

6.4.3. Managing the video stream

6.4.3.1. Managing the video stream connection security

Since the client-side Kubernetes *job* runs the same docker image seen in the docker implementation, the video stream connection security mechanism is actually the same shown in section 6.3.3.1.. In fact, to implement it, the *ssh* command will be used to create a tunnel with the *ssh port forwarding mechanism*, which will wrap the video stream.

Indeed, the difference is in the way the parameter will be passed to the script inside the container and the volumes will be mounted in it. This is because, of course, in this case no *docker run* command will be used but the *vncviewer.yaml* file, in which the *job* itself is defined, will be applied.

The *ssh* command, as seen in section 6.3.3.1., requires:

- the *ssh private key*, contained in a host's machine file that will be mounted inside the container;
- the *run_vncviewer.sh* script *enc* execution parameter, a boolean that specifies whether the encryption has been enabled;
- the *run_vncviewer.sh* script *enc_port* execution parameter, which specifies the VNC server port;
- the *run_vncviewer.sh* script *target_node_ip* execution parameter, which should be the remote node IP address. We will see afterwards that something different will be used;
- the *run_vncviewer.sh* script *target_node_port_ssh* execution parameter, which specifies the SSH server port;

The *vncviewer.yaml* code snippet below will show how the mentioned file containing the *ssh private key* will be mounted inside the container and how the required parameters will be passed to the container entry point.

```
apiVersion: batch/v1
kind: Job
[...]
spec:
  template:
    [...]
    spec:
      [...]
      containers:
        - name: RAR_002_TIGERVNC_CONTAINER_NAME
          [...]
          volumeMounts:
            - mountPath: /home/vnc/ssh_id_rsa
              name: workingdir
            [...]
          args:
            - "RAR_004_ENC"
            [...]
            - "RAR_010_TARGET_NODE_IP"
            - "RAR_011_TARGET_NODE_PORT_SSH"
            - "RAR_012_ENC_PORT"
            [...]
      volumes:
        - name: workingdir
          hostPath:
            path: RAR_015_WORKING_DIR
        [...]
```

As you can see in the code snippet above, since the *run_vncviewer.sh* script is the container *entrypoint*, we can pass the required parameters through the *yaml args* field as a list of arguments. Of course, since the *vncviewer.yaml* file acts as a template similarly to the *deployment.yaml* file seen in the server-side implementation, the arguments are parametrized and will be replaced with their values from the *cloudify* script as shown in the code snippet below.

```
function adjust_and_apply_pod {
    [...]
    sed -i "s#RAR_004_ENC#enc=$enc#g" $tigervncPodFile
    [...]
    sed -i "s#RAR_010_TARGET_NODE_IP#target_node_ip=$targetNodeIp#g" \
       $tigervncPodFile
    sed -i "s#RAR_011_TARGET_NODE_PORT_SSH#target_node_port_ssh=$targetNodePortSsh#g" \
       $tigervncPodFile
    sed -i "s#RAR_012_ENC_PORT#enc_port=$port#g" $tigervncPodFile
    [...]
    sed -i "s#RAR_015_WORKING_DIR#$working_dir#g" $tigervncPodFile
    [...]
}
```

As you can see above, each *RAR*_***_***** parameter will be replaced, as expected by the *run_vncviewer.sh* script, with a string like this one <*param_name*>=<*param_value*>.

Note that, even if the *$enc* variable is actually the same seen in section 6.3.3.1., the other variables will have different values. In fact, since the remote node's IP address is no longer required and the *ClusterIP service* has been used in place of the *NodePort*, they have been set as shown in the *cloudify* script code snippet below.

```
function prepare_env_runmode_2 {
         port=5900
         [...]
         targetNodeIp=$svc_URL
         targetNodePortSsh=$ssh_port
}
```

So, the *$port* variable value will be hard-coded because it will be always the same since only the VNC protocol has been implemented, the *$targetNodeIp* has been replaced with the *service URL* retrieved as shown in section 6.4.2., and the *$targetNodePortSsh* will be hard-coded too because it will be always the same since no *NodePort* service will be used this time and just the *SSH* server port will be exposed.

Finally, a tiny specification about the *ssh private key* file mounting should be done. In fact, mounting a local filesystem file is possible because, as we will see afterwards, the client-side pod will be always scheduled on the local node. Otherwise this solution wouldn't have worked. Moreover, the *RAR_015_WORKING_DIR* parameter will be replaced with the *ssh private key* file path.

6.4.3.2. Managing the VNC video stream

Similarly to the previous section, since the client-side Kubernetes *job* runs the same docker image seen in the docker implementation, the VNC video stream management is actually the same shown in section 6.3.3.2.. In fact, to implement it, the *vncviewer* command will be used to connect to the VNC server.

Even in this case, the difference is in the way the parameter will be passed to the script inside the container, the volumes will be mounted, and the environment variables will be defined in it. This is because, as said before, in this case no *docker run* command will be used but the *vncviewer.yaml* file will be applied. As first, we will analyze the *vncviewer* command parameter passing, then we will discuss about the volumes to mount and the environment variable to expose.

The *vncviewer* command, as seen in section 6.3.3.2., requires:

- the *run_vncviewer.sh* script *compression* execution parameter, which specifies the compression level;
- the *run_vncviewer.sh* script *quality* execution parameter, which specifies the image quality;
- the *run_vncviewer.sh* script *target* execution parameter, which is a string composed as follows *<target_IPaddress>::<VNC_port>* (see section 6.2.3.2.). Unlike the native and the containerized implementations, in which an IP addresses will be actually used, in this case it will be replaced by the *svc_URL cloudify* script variable seen in section 6.4.2.. So, the *$target* variable will be obtained as shown in the *cloudify* code snippet below.

```
function prepare_env_runmode_2 {
     [...]
     if [ $enc -eq 1 ]; then
          echo "Starting encrypted VNC connection..."
          target="localhost::$port"
     else
          echo "Starting clear VNC connection..."
          target="$svc_URL::$port"
     fi
     [...]
}
```

  As you can see, the *if-then-fi* statement above is exactly the same we have seen in section 6.2.3.2. for the native implementation except for: the *$targetNodeIp*, which has been replaced with the *$svc_URL* variable, and the *$targetNodePortProtocol*, which has been replaced with the *$port* since variable discussed in section 6.4.3.1.;
- the *run_vncviewer.sh* script *token* execution parameter, which is used to allow the client to access the VNC server.

The *vncviewer.yaml* code snippet below will show how the required parameters will be passed to the container entry point.

```
apiVersion: batch/v1
kind: Job
[...]
spec:
  template:
    [...]
    spec:
      [...]
      containers:
        - name: RAR_002_TIGERVNC_CONTAINER_NAME
          [...]
          args:
            [...]
            - "RAR_006_COMPRESSION"
            - "RAR_007_QUALITY"
            - "RAR_008_TARGET"
            - "RAR_009_TOKEN"
            [...]
```

As you can see in the code snippet above, similarly to the previous section, since the *run_vncviewer.sh* script is the container *entrypoint*, we can pass the required parameters through the *yaml args* field as a list of arguments. Even in this case, the arguments are parametrized and will be replaced with their values from the *cloudify* script as shown in the code snippet below.

```
function adjust_and_apply_pod {
   [...]
   sed -i "s#RAR_006_COMPRESSION#compression=$compression#g" $tigervncPodFile
   sed -i "s#RAR_007_QUALITY#quality=$quality#g" $tigervncPodFile
   sed -i "s#RAR_008_TARGET#target=$target#g" $tigervncPodFile
   sed -i "s#RAR_009_TOKEN#token=$token#g" $tigervncPodFile
   [...]
}
```

As you can see above, even in this case each *RAR*_***_***** parameter will be replaced, as expected by the *run_vncviewer.sh* script, with a string like this one *<param_name>*=*<param_value>*. Note that, except for the *$target* variable discussed above, the others are exactly the same seen in section 6.3.3.2..

Finally, now that we have seen how the *vncviewer* parameters will be passed to the *run_vncviewer.sh* script, we can focus our attention on the required volumes to be mounted and the environment variable to be exposed in the local *pod* to enable the *vncviewer* to show the Graphical User Interface work properly. As we have seen in section 6.3.3.2., we refer to the following volumes/environment variables:

- the *shared memory* (*/dev/shm*) volume, to avoid crashes;
- the host's *D-Bus* (/dev/dbus) volume, an inter-process communication mechanism required to avoid crashes too;
- the host's */tmp/.X11-unix/* volume, contains the video Unix-domain socket;
- the *DISPLAY* environment variable, that selects which video Unix socket should be used. Combined with the */tmp/.X11-unix/* allows the container to exploit the host's video interface.

The *vncviewer.yaml* code snippet below will show how they will be mounted/exposed inside the *pod.*

```
apiVersion: batch/v1
kind: Job
[...]
spec:
  template:
    [...]
    spec:
      [...]
      containers:
        - name: RAR_002_TIGERVNC_CONTAINER_NAME
          [...]
          env:
            - name: DISPLAY
              value: :0
          [...]
          volumeMounts:
            [...]
            - mountPath: /tmp/.X11-unix
              name: video
            [...]
            - mountPath: /dev/shm
              name: tempsystem
            - mountPath: /var/run/dbus
              name: systembus
          [...]
      volumes:
        [...]
        - name: video
          hostPath:
            path: /tmp/.X11-unix
        [...]
        - name: tempsystem
          hostPath:
            path: /dev/shm
        - name: systembus
          hostPath:
            path: /var/run/dbus
      [...]
```

This way all requirements are met, so the GUI will be displayed correctly.

6.4.4. Managing the audio stream

Similarly to the previous sections, since the client-side Kubernetes *job* runs the same docker image seen in the docker implementation, the audio stream management is actually the same shown in section 6.3.4.. In fact, to implement it, the PulseAudio server have to be launched and enabled to receive the audio stream through a TCP connection wrapped in an *ssh remote port forwarding* tunnel.

The difference is in the way the sound device will be mounted. This is because, as said before, in this case no *docker run* command will be used but the *vncviewer.yaml* file will be applied. So, to make the audio work properly we just need to mount the host's sound card device inside the container as shown in the *vncviewer.yaml* code snippet below.

```
apiVersion: batch/v1
kind: Job
[...]
spec:
  template:
    [...]
    spec:
      [...]
      containers:
        - name: RAR_002_TIGERVNC_CONTAINER_NAME
          [...]
          volumeMounts:
            [...]
            - mountPath: /dev/snd
              name: sound
            [...]
          securityContext:
            privileged: true
          [...]
      volumes:
        [...]
        - name: sound
          hostPath:
            path: /dev/snd
```

As you can see in the code snippet above, unlike the docker implementation we have seen in section 6.3.4. in which the sound card has been mounted as a device through the *docker run --device* parameter, Kubernetes doesn't have an equivalent mounting feature so, we are forced to mount it as a volume. However, this way it will not work because the container does not recognize it as a device and the PulseAudio daemon will not load the sound card during its initialization process. Anyway, we can solve this problem by specifying a privileged *securityContext* field. This way the PulseAudio daemon will recognize the sound card and the sound will be reproduced correctly.

Finally, the *ssh* command that will create the *remote port forwarding* tunnel requires the following elements:

- the *ssh private key*, contained in a host's machine file that will be mounted inside the container;
- the *run_vncviewer.sh* script *enc* execution parameter, a boolean that specifies whether the encryption has been enabled;
- the *run_vncviewer.sh* script *enc_port* execution parameter, which specifies the VNC server port;
- the *run_vncviewer.sh* script *target_node_ip* execution parameter, which should be the remote node IP address but, in this case, as seen before, is the service URL;
- the *run_vncviewer.sh* script *target_node_port_ssh* execution parameter, which specifies the SSH server port.

These elements have been retrieved as seen in section 6.4.3.1..

6.4.5. Managing the application termination

In the previous sections we said that all the architecture lifecycle is managed through the *cloudify* script which is in charge of deploy the server-side and the client-side and once the target application execution ends it will destroy them and cleanup all the allocated resources.

In the other implementations, the remote application termination has been caught by the *cloudify* script in these ways:

- in the native implementation, since the *vncviewer* is a blocking command, once the viewer will be closed that command will return and the *cloudify* script resume its execution;
- in the docker implementation will be used the *docker wait <container_name>* blocking command which will return once the docker execution has been completed, then the *cloudify* script can resume its execution.

In the Kubernetes implementation, actually, there is the command *Kubectl wait* which has the same *docker wait* purpose. What we expect to do is to catch the *job Completed* status to clean up all the resources. This is one of the reasons why we choose to use a *job* to implement the Kubernetes server-side. Unfortunately, the *kubectl wait* command has been marked *"Experimental"* and it doesn't have always the expected behavior. For this reason, we need to implement an alternative system to be sure the *job* termination will be caught.

We can have two solutions:

1. implementing a polling system in which the *job* status will be checked every *n* seconds and, once it becomes *"Completed"*, the *cloudify* script will resume its execution;
2. implementing a signaling system in which is the *job*-related *pod* that tells the *cloudify* script that the execution is over.

Since the first solution can generate some overhead in terms of CPU consumption and lack of responsiveness, we decided to implement the second one.

For this purpose, the *cloudify* script uses a *netcat* server which will wait for a remote signal and the *run_vncviewer.sh* script will use a *netcat* client which will send a message once its process is going to terminate.

6.4.5.1. Implementing the server-side signaling system

To realize the signaling system above, we need to prepare the *netcat* server environment in the *cloudify* script then to execute the *netcat* server itself. The first thing to is to find a free TCP port in the host.

```
function adjust_and_apply_pod {
  [...]
  while true; do
    pod_wait_port=$(shuf -i 1025-65535 -n 1)

    read -r pod_port_to_check_local_ip <<< \
       $(nc -zv localhost $pod_wait_port 2>&1)
    read -r pod_port_to_check_localhost <<< \
       $(nc -zv $current_node_address $pod_wait_port 2>&1)
    #Check generated port
    if [[ $( echo $pod_port_to_check_local_ip | grep succeeded) == "" ]] && \
       [[ $( echo $pod_port_to_check_localhost | grep succeeded) == "" ]]; then
      #It's a free port. OK
      break;
    fi
  done
  [...]
}
```

As you can see in the *cloudify* code snippet above, since the first 1024 ports are reserved, a number between 1025 and 65535 will be generated by executing the *shuf* command. Then, through the *netcat -zv* command it will be checked whether that is a free port or not. The loop will be repeated until a free port will be found.

Once a free port has been found, we can execute the *netcat* server.

```
function adjust_and_apply_pod {
  [...]
  pod_finished_can_exit=0
  while [ $pod_finished_can_exit -eq 0 ]; do
    while read line; do
      if [ "$line" == "Close_${token}" ]; then
           pod_finished_can_exit=1
           break
      fi
    done < <(nc -q -1 -l $pod_wait_port)
  done

}
```

The *cloudify* script code snippet above shows how the signaling system server-side has been implemented. The actual server will be launched through th *nc* command which will receive the following parameters:

- *-q* -1, which will force the *nc* command not to end once it receives the EOF on *stdin*. This behavior is due to the negative number specification (-1) for the *-q* parameter and it will prevent the premature termination of the *nc* server since we need to wait forever until a message will be received;
- *-l*, which specifies that the *nc* process have to wait for connection on the specified port. In our case the port is the one specified in the varible *$pod_wait_port* and retrieved as seen in the previous code snippet above.

Once the *nc* process will receive a connection request it will wait for a message that will be redirected on the *stdin* and stored in the *$line* variable through the *read* command. Then, the script will check that the message corresponds to the expected one and, since it means that the remote application execution is over, it will clean up all the resources and ends. Note that the script will wait for the right message before exit to prevent that some attacker forces the resources cleanup before the actual application termination. Moreover, for this purpose, the message contains the One Time Token generated, as we have seen in section 5.3.2., for the VNC server access and stored in the *${token}* variable, so that making attacks will be even more difficult than a static message.

6.4.5.2. Implementing the client-side signaling system

The client-side signaling system, of course, has been implemented inside the *run_vncviewer.sh* script.

```
function close_connection {
nc ${!client_host_ip} ${!client_host_port} <<-EOF 1>&2
    Close_${!token}

EOF
}

function clean_and_exit {
    [...]
    if [[ ${!pod} -eq 1 ]]; then
        echo "Send pod terminating status to cloudify..." 1>&2
        close_connection
        echo "Terminating sent." 1>&2
    fi
    [...]
}
```

The *run_vncviewer.sh* code snippet above shows how it has been implemented. As you can see, once the script execution is going to terminate, it will check if the *run_vncviewer.sh pod* execution parameter is set to 1. We have not discussed about this yet, but this is a mandatory parameter which tells the script how to behave once it is going to terminate its execution. Since the docker image is the same both in the docker run mode and in the pod run mode, the script needs to know whether it is the first (*pod=0*) or the second (*pod=1*) case just because the second one requires to signal the process termination to the *cloudify* script.

Then, the *close_connection* function will be executed. This one is in charge of sending the closing message (which also contains the One Time Token as discussed in the previous section) through the *nc* command. As you can see, the closing message is not a simple string but has two lines: this is because the *ns* server requires the carriage return as string termination, otherwise it will be stuck forever waiting for the message.

A part the *run_vncviewer.sh* execution parameter named *pod*, also the following one are mandatory in this case:

- *client_host_ip,* which specifies the IP address of the host running the *cloudify* script;
- *client_host_port*, which specifies the port the *nc* server is listening to, generated as shown in the previous section.

The three parameters above, as we have seen for the other *run_vncviewer.sh* parameters in the previous sections, will be passed through the vncviewer.*yaml args* field as a list of arguments. The code snippet below will show it.

```
apiVersion: batch/v1
kind: Job
[...]
spec:
  template:
    [...]
    spec:
      [...]
      containers:
        - name: RAR_002_TIGERVNC_CONTAINER_NAME
          [...]
          args:
            [...]
            - "RAR_005_POD"
            [...]
            - "RAR_013_CLIENT_HOST_IP"
            - "RAR_014_CLIENT_HOST_PORT"
          [...]
```

Even in this case, the arguments are parametrized and will be replaced with their values by the *cloudify* script as shown in the code snippet below.

```
function adjust_and_apply_pod {
   [...]
   sed -i "s#RAR_005_POD#pod=1#g" $tigervncPodFile
   [...]
   sed -i "s#RAR_013_CLIENT_HOST_IP#client_host_ip=$current_node_address#g" \
      $tigervncPodFile
   sed -i "s#RAR_014_CLIENT_HOST_PORT#client_host_port=$pod_wait_port#g" $tigervncPodFile
   [...]
}
```

As you can see above, even in this case each *RAR_****_***** parameter will be replaced, as expected by the *run_vncviewer.sh* script, with a string like this one *<param_name>=<param_value>*.

We have discussed the *$pod_wait_port* variable and the *pod* execution parameter before but we havent't talked about the *$current_node_address* variable yet. It contains the local machine IP address and will be retrieved as shown in the *cloudify* script code snippet below.

```
function retrieve_curr_k8s_node_info {
  [...]
  IFS='#' read -r -a current_node_info <<< $(kubectl get nodes -l
kubernetes.io/hostname=`cat /etc/hostname` -o 'jsonpath={range
.items[0].status.addresses[*]}{.type}={.address}#{end}')
  for ((i=0; i<${#current_node_info[@]}; i++)); do
    [...]
    if [[ $(echo ${current_node_info[$i]} | cut -d '=' -f 1) == "InternalIP" ]];then
      current_node_address=$(echo ${current_node_info[$i]} | cut -d '=' -f 2)
    fi
  done
  [...]
}
```

As you can see, the function used above is actually the one we have seen in section 5.3.5. where we wanted to retrieve the node name of the local machine. The mechanism, even in this case, is the same used in the mentioned section but this time we will retrieve the IP address bound to the node which has the name equals to the one written in the */etc/hostname* file.

6.4.6. Managing the pod scheduling

We have discussed, so far, how the audio/video remotization has been implemented with pod run mode but there is another important challenge since, as mentioned in section 4.4.4., this way we run the risk of executing the VNC viewer *pod* in a node that is not the local one. This is an undesirable event because even if we want to offload the desktop application execution, we still want to interact with it from the local machine. To achieve this goal, we needed to make a tiny customization in the *vncviewer.yaml* file.

```
apiVersion: batch/v1
kind: Job
[...]
spec:
  template:
    [...]
    spec:
      nodeName: RAR_001_K8S_NODE_NAME
      [...]
```

As you can see in the code snippet above, inside the *pod specs* we specified the *nodeName* which is used to force the *pod* scheduling and execution in the node having the specified name.

Even in this case, the node name is parametrized and will be replaced with its value by the *cloudify* script as shown in the code snippet below.

```
function adjust_and_apply_pod {
    [...]
    sed -i "s#RAR_001_K8S_NODE_NAME#$current_node_name#g" $tigervncPodFile
    [...]
}
```

The *$current_node_name* variable used above is the same retrieved, as shown in section 5.3.5., through the *cloudify* script function *retrieve_curr_k8s_node_info* by using the */etc/hostname* file content, and represents the local machine node name. In the case discussed in the mentioned section that has been used to avoid the remote *pod* scheduling on the local machine, in this case it is used exactly for the opposite purpose.

6.4.7. Managing the applications concurrency

In the Kubernetes server-side implementation we have seen that since a *deployment* is used to create the server-side *pod*, if we want to run more instances of the same application, we need to parametrize the resources name to uniquely identify them. Actually, even in this case, we need to use a different *job* name and "app" *label* to uniquely identify the *job* and the related *pod*, and to gracefully clean up the allocated resources once the execution ends. For this purpose, we parametrized their values as shown in the following *vncviewer.yaml* code snippet.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: RAR_000_APP_NAME_PID-job
  labels:
    app: RAR_000_APP_NAME_PID
spec:
  template:
    metadata:
      labels:
        app: RAR_000_APP_NAME_PID
    spec:
      [...]
      containers:
        - name: RAR_002_TIGERVNC_CONTAINER_NAME
          [...]
```

As you can see above, the *job* name, the "*app*" *selector* and the *pod* "app" *label* have been parametrized through the *RAR_000_APP_NAME_PID* string. Note that the *job* name is actually composed by a variable part followed by the "-job" static string.

Even the *pod* container name has been parametrized but with the *RAR_002_TIGERVNC_CONTAINER_NAME* string.

As usual, the mentioned parameters will be replaced with their values by the *cloudify* script as shown in the code snippet below.

```
function adjust_and_apply_pod {
    [...]
    sed -i "s#RAR_000_APP_NAME_PID#$vnc_cli_app_pid#g" $tigervncPodFile
    [...]
    sed -i "s#RAR_002_TIGERVNC_CONTAINER_NAME#$vnc_cli_app_pid#g" $tigervncPodFile
    [...]
}
```

The *$vnc_cli_app_pid* variable is the same we have seen in section 6.3.5. and is composed by a static part, which is a string hard coded inside the $*tigervnc_container_name* variable, and a PID which has been generated as shown in section 5.3.8. and stored in the *$app_pid* variable.

Chapter 7 - Validation

7.1. Supported applications and exploited hardware capabilities

As mentioned in section 5.2.1.2., the *app_image* Dockerfile has been realized to be used as a sort of template to install the required target application inside the docker image. In fact, since it uses the Debian *apt-get* package manager, basically all the applications provided with that kind of installation package can be installed.

Moreover, to enable the user to install application that are not available on the chosen Linux Distribution official repositories, the REPO_TO_ADD *docker build arg* has been provided.

So far, the supported applications are:

- Mozilla Firefox, a widely used Internet browser;
- Libreoffice, a free and powerful office suite;
- Blender, a «free and open source 3D creation suite. It supports the entirety of the 3D pipeline-modeling, rigging, animation, simulation, rendering, compositing and motion tracking, video editing and 2D animation pipeline»[50].

Those applications have been installed using "*ubuntu:18.04*" as a base image and widely tested to check they work properly in each client-side run mode.

Furthermore, installing Blender inside the Docker image required the addition of the "*ppa:thomas-schiex/blender*" source repository since on the chosen base image only the *snap* package is available.

In general, the application offloading enables the user to exploit all the remote hardware capabilities. It means that also the remote graphic card may be used, if there are any. So, since Blender is an open source 3D creation suite, a Docker image exploiting the remote graphic card capability has been created. For this purpose, assuming that the remote host is provided with a NVIDIA graphic card, a Blender image has been created by using the *"nvidia/cuda:10.2-runtime-ubuntu18.04"* base image. This one is actually built on the "*ubuntu:18.04*" parent image by adding the required NVIDIA libraries. This way the base image will provide the same environment obtained for the other images created by using the *"ubuntu:18.04"*, and by adding the libraries required to exploit the remote NVIDIA graphic card capabilities. Actually, using the aforementioned base image is not sufficient. In fact, the remote node must be provided with:

- NVIDIA CUDA drivers, natively installed on the remote operating system;
- *nvidia-container-runtime*, «a GPU aware container runtime, compatible with the Open Containers Initiative (OCI) specification used by Docker, CRI-O, and other popular container technologies. It simplifies the process of building and deploying containerized GPU-accelerated applications to desktop, cloud or data centers. With NVIDIA Container Runtime supported container technologies like Docker, developers can wrap their GPU-accelerated applications along with its

[50] Blender official website - About, https://www.blender.org/, October 12th, 2020.

dependencies into a single package that is guaranteed to deliver the best performance on NVIDIA GPUs, regardless of the deployment environment»[51].

The Blender image created above has been successfully tested by scheduling the related pod on a Kubernetes node provided with a NVIDIA graphic card along with the required drivers, packages and configurations. A 2D image rendering has been executed by using both a basic Blender image and a GPU aware Blender image. The latter image execution has shown a sensitive reduction of the execution time due to the GPU exploitation.

7.2. Measures

7.2.1. Resource consumption: native vs containerized vs cloudified application

So far, the supported applications and the related hardware capability exploitation have been discussed. This section will present a comparison between the resource consumption data of one of the supported native applications and the related containerized implementation. For this purpose, since the client-side is provided with three run modes (native, Docker container and Kubernetes pod) the focus will be placed on both the server-side and the client-side pointing out how the resource consumption on the local machine changes compared to the native application execution. For this purpose, the Firefox image has been chosen.

7.2.1.1. Execution context specifications

The tests have been made on a vanilla Kubernetes cluster composed by four nodes. Each node is a VirtualBox virtual machine with the following hardware and software specifications:

Node	**Specifications**		
Master	System	Base memory	6144 MB
		Processors	2
		Acceleration	VT-x/AMD-V, Nested Paging, KVM Paravirtualization
	Video	Video memory	128 MB
		Graphics controller	VMSVGA
		Acceleration	3D
	Operating system	Ubuntu Server 20.04 LTS	
	Docker	Version 19.03.12	
	Kubernetes	Version 1.18.5	

[51] Nvidia developer official website - NVIDIA Container Runtime, https://developer.nvidia.com/nvidia-container-runtime, October 12th, 2020.

Node	Specifications		
Worker-01	System	Base memory	6144 MB
		Processors	1
		Acceleration	VT-x/AMD-V, Nested Paging, KVM Paravirtualization
	Video	Video memory	128 MB
		Graphics controller	VMSVGA
		Acceleration	3D
	Operating system	Ubuntu Server 20.04 LTS	
	Docker	Version 19.03.12	
	Kubernetes	Version 1.18.5	

Node	Specifications		
Worker-02	System	Base memory	6144 MB
		Processors	1
		Acceleration	VT-x/AMD-V, Nested Paging, KVM Paravirtualization
	Video	Video memory	128 MB
		Graphics controller	VMSVGA
		Acceleration	3D
	Operating system	Ubuntu Server 20.04 LTS	
	Docker	Version 19.03.12	
	Kubernetes	Version 1.18.5	

Node	Specifications		
Client-01	System	Base memory	6144 MB
		Processors	2
		Acceleration	VT-x/AMD-V, Nested Paging, KVM Paravirtualization
	Video	Video memory	128 MB
		Graphics controller	VMSVGA
		Acceleration	3D
	Operating system	Ubuntu Desktop 20.04 LTS	

	Docker	Version 19.03.12
	Kubernetes	Version 1.18.5

The physical host is provided with the following hardware and software:

- CPU: Intel Core i7-6820HK;
- RAM: 32GB;
- Graphic card: NVIDIA GeForce GTX 980M;
- Operating system: Microsoft Windows 10;
- VirtualBox version: 6.1.12 r139181.

7.2.1.2. Measure specifications and used tools

The tests have been made by executing the *cloudify* script with the following settings:

- VNC video quality: 5;
- VNC video compression: 2;
- Secure connections: enabled.

Note that the *cloudify* script has been executed always in the *Client-01* node since it is the only one provided with a desktop interface. Besides, the server-side has been scheduled always in the *Master-01* node. To obtain this, the scheduling on the other nodes has been disabled by adding a *taint* on each through the following command execution:

kubectl taint nodes <node_name> key=value:NoSchedule

Finally, disabling the scheduling on the *Client-01* node was not necessary since, as seen in section 5.3.5., the cloudify script force the server-side pod execution on a foreign node by construction.

For each measure, has been reproduced always the same video in streaming with Firefox, following always the same steps to find and reproduce it, and paying attention to specify always the same image quality (720p) in order to obtain significant values for each measured resource.

The following sections will report the measured values of:

- CPU, expressed in percentage of usage on the local machine and in milli-units on the remote node. Note that the milli-units express the number of requests per seconds. For example, 500m would be half a request per second, 10000m would be 10 requests per second, and 10500m would be 10.5 requests per second;
- RAM, expressed in MiB;
- Network traffic transmitted in bytes;
- Network traffic received bytes.

The tables in the following sections will contain the average value for each resource, obtained by executing the arithmetic mean over a sample set. For this purpose, a sample rate of 1 second for each measure has been chosen. Finally, to make that values statistically relevant, the measures have been made three times, one for each application execution.

To obtain the samples described above, several tools have been employed:

- *top*, used to retrieve the CPU usage in percentage and the memory consumed in bytes of a native process;
- *nethogs*, shows the transmitted/received network traffic in bytes of a native process;
- *docker stats*, returns the CPU usage in percentage, the memory consumed in bytes and the transmitted/received network traffic in bytes of a Docker container;
- *kubectl top*, returns the CPU usage in milli-units and the memory consumed in bytes of a Kubernetes pod;
- *Prometheus*, is a Kubernetes monitoring system. In this case, it has been used to retrieve the transmitted/received network traffic in bytes of a Kubernetes pod.

7.2.1.3. Firefox native execution

The first measures to report are those related to the native Firefox execution. Note that, in this case the *top* and the *nethogs* tools have been used to retrieve the sample values.

The first tool allows the process monitoring by specifying its PID. Since Firefox during its execution creates several child processes, a script has been created to collect all the PIDs related to Firefox and to retrieve the CPU and memory values for all of them.

```bash
#!/bin/bash

logfile="./measures/firefox_cpu_mem_03.txt"

#Init logfile
echo "" > $logfile

while true; do
    #Init pid array
    pids=()

    #Set line separator
    IFS=$'\n'

    #Retrieve all pid matching firefox process
    for line in $(ps aux | grep firefox); do
        val=$(echo $line | cut -d " " -f 3)
        if [[ $val -ne "" ]]; then
            #Add pid to pid array
            pids+=("-p$val")
        fi
    done

    #Run top with all the arrays
    top -b -n1 ${pids[@]} >> $logfile

    #Sleep
    sleep 1
done
```

The obtained values for each process will be added together to obtain the total amount for each resource.

The *nethogs* tool allows to continuously monitor the network activity of a process. It shows the process name, its PID, and the total amount of bytes transmitted and received since the target process start. Actually, if the *nethogs* application is launched during the target application execution, it will collect only the values from that moment ahead. For this reason, *nethogs* has been launched before the target application execution. Once the target application ends, it is possible to retrieve its total amount of network traffic.

The table below shows the average values of each resource.

Measure nr.	CPU usage (%)	RAM usage (MiB)	Network transmitted traffic (MiB)	Network received traffic (MiB)
1	52,01%	1005,33 MiB	1,614 MiB	44,920 MiB
2	64,69%	1055,24 MiB	1,510 MiB	45,788 MiB
3	48,57%	999,65 MiB	1,614 MiB	43,266 MiB
	55,09%	1020,07 MiB	1,579 MiB	44,658 MiB

Note that the last line contains the average values among the three measures for each resource.

7.2.1.4. Firefox execution with client-side native run mode

This section is related to the execution of the target application on a remote pod in the case the vncviewer, the ssh client and the PulseAudio server have been installed natively on the local host. In this case, the measures of both the server-side pod and the client-side processes will be shown. For this purpose, the *top* and the *nethogs* tools have been used for the native processes monitoring, while the *kubectl top* and the *Prometheus* tools have been used for the remote pod monitoring.

To be more specific, since the client-side is composed by both the vncviewer and the ssh tools, both will be monitored. The obtained values for each process will be added together to obtain the total amount for each resource. Actually, in the client-side there is also the PulseAudio process but since it is a daemon always running on the operating system, its resources consumption can be ignored. Finally, since the *secure connection* option has been enabled, the network traffic related to both the audio and the video streaming will be received and transmitted only by the *ssh* process.

The table below shows the average values of each resource for the client-side.

Measure nr.	CPU usage (ssh% + vncviewer%)	RAM usage (ssh MiB + vncviewer MiB)	Network transmitted traffic (MiB)	Network received traffic (MiB)
1	(2,4 + 27,53) %	(3,55 + 14,62) MiB	3,610 MiB	190,636 MiB
2	(2,45 + 31,77) %	(3,48 + 14,57) MiB	3,873 MiB	215,848 MiB
3	(2,55+ 31,98) %	(3,48 + 14,63) MiB	3,983 MiB	221,564 MiB
	32,89%	18,11 MiB	3,821 MiB	209,350 MiB

Note that the last line contains the average values among the three measures for each resource.

The table below shows the average values of each resource for the server-side.

Measure nr.	CPU usage (milli-units)	RAM usage (MiB)	Network transmitted traffic (MiB)	Network received traffic (MiB)
1	728,28m	532,18 MiB	190,60 MiB	59,34 MiB
2	548,00m	426,22 MiB	210,00 MiB	69,46 MiB
3	716,02m	513,99 MiB	186,07 MiB	63,68 MiB
	664,10m	490,80 MiB	195,56 MiB	64,16 MiB

Note that the last line contains the average values among the three measures for each resource.

Analyzing the values shown in the tables above, it can be seen that the resources consumption in the client-side is much lower than the native application one. Moreover, even if it is not possible to compare the CPU usage since *kubectl top* command provides it only in milli-units, while *top* command provides it only in percentage, it is possible to see that basically there is not a resource consumption worsening due to the application containerization in the server-side. Actually, just the network usage increased. In fact, the received traffic from the client-side, as well as the transmitted traffic from the server-side, is higher than the received traffic of the native application. This overhead is due to the VNC and audio streaming. Anyway, if the network bandwidth is sufficiently wide, this overhead may be acceptable considering the benefits due to the remote node hardware capabilities exploitation. In addition, it is possible to act on the VNC video compression and quality to decrease the bandwidth consumption if needed.

7.2.1.5. Firefox execution with client-side Docker container run mode

This section is related to the execution of the target application on a remote pod in the case the client-side has been containerized too and executed through the *docker run* command.

In this case, as in the previous one, the measures of both the server-side pod and the client-side processes will be shown. For this purpose, the *kubectl stats* tool has been used for the client side monitoring, and and the *kubectl top* and the *Prometheus* tools have been used for the remote pod monitoring.

The table below shows the average values of each resource for the client-side.

Measure nr.	CPU usage (%)	RAM usage (MiB)	Network transmitted traffic (MiB)	Network received traffic (MiB)
1	43,76%	11,69 MiB	4,04 MiB	243,19 MiB
2	44,56%	11,38 MiB	4,06 MiB	244,14 MiB
3	43,80%	11,77 MiB	3,99 MiB	232,70 MiB
	44,04%	11,61 MiB	4,03 MiB	240,01 MiB

Note that the last line contains the average values among the three measures for each resource.

The table below shows the average values of each resource for the server-side.

Measure nr.	CPU usage (milli-units)	RAM usage (MiB)	Network transmitted traffic (MiB)	Network received traffic (MiB)
1	655,3m	530,82 MiB	216,62 MiB	63,32 MiB
2	499,06m	411,24 MiB	222,59 MiB	66,85 MiB
3	553,71m	416,35 MiB	230,91 MiB	70,92 MiB
	569,36m	452,80 MiB	223,37 MiB	67,03 MiB

Note that the last line contains the average values among the three measures for each resource.

Analyzing the values shown in the tables above, it can be seen that the resource consumption in the client-side, except for the network traffic, is lower than the native Firefox application execution. Compared to the docker container run mode, the containerized client-side resource consumption is very similar to the native one. Finally, the server-side pod resource consumption is very similar to the that shown in the previous section.

Also in this case, the increasing of the network traffic is due to the VNC and audio streaming. Thus, the same observations can be made.

7.2.1.6. Firefox execution with client-side Kubernetes pod run mode

This section is related to the fully cloudified exexution. In this case, as in the previous one, the measures of both the server-side pod and the client-side processes will be shown. For this purpose, since all the execution has been cloudified, just the *kubectl top* and the *Prometheus* tools will be necessary. Anyway, in order to provide an estimation on the CPU percentage consumption of the client-side pod, the *docker stats* tool has been used to monitor the container executed in the pod itself.

The table below shows the average values of each resource for the client-side.

Measure nr.	CPU usage (m and %)	RAM usage (MiB)	Network transmitted traffic (MiB)	Network received traffic (MiB)
1	260,14m 17,73%	20,13 MiB	3,46 MiB	188,27 MiB
2	178,66m 35,64%	10,27 MiB	3,61 MiB	187,52 MiB
3	334,37m 39,10%	10,24 MiB	3,43 MiB	184,30 MiB
	257,72m 30,82%	13,55 MB	3,50 MiB	125,26 MiB

Note that the last line contains the average values among the three measures for each resource.

The table below shows the average values of each resource for the server-side.

Measure nr.	CPU usage (milli-units)	RAM usage (MiB)	Network transmitted traffic (MiB)	Network received traffic (MiB)
1	497,34m	403,51 MiB	213,83 MiB	61,88 MiB
2	709,46m	489,19 MiB	200,89 MiB	63,89 MiB
3	713,06m	467,14 MiB	226,20 MiB	60,92 MiB
	639,95m	453,28 MiB	213,64 MiB	62,23 MiB

Note that the last line contains the average values among the three measures for each resource.

Analyzing the values shown in the tables above, including in this case it can be seen that the resource consumption in the client-side, except for the network traffic, is lower than the native Firefox application execution. Compared to the other run modes, the client-side resource consumption is very similar. This was an expected behavior since the pod executes a Docker container too. Finally, the server-side pod resource consumption is very similar to that shown in the other client-side run modes.

The increase of the network traffic, as for the other cases, is due to the VNC and audio streaming. Thus, the same observations can be made.

7.2.2. Start-up time consumption analysis

So far, the resource consumption in each kind of execution mode has been analyzed. Actually, focusing on the cases in which the server-side have been containerized and remotized on a remote node, there will be a startup time consumption overhead. This is because the Kubernetes cluster requires some time to schedule and start the pod on the remote node. In the following sections will be shown how the startup time changes in each kind of execution, highlighting how much time will be spent on the local host and on the foreign node until the target application starts.

7.2.2.1. Firefox execution with client-side native run mode

In this execution mode, the *cloudify* script will prepare at first the *deployment.yaml* file according to the execution parameters, then will apply the Deployment. The time between the beginning of the execution and the *kubectl apply* command has been named T1. Then, the *cloudify* script will wait for the pod to be in *Running* status. The time between the *kubectl apply* command and the pod *Running* status has been named T2 ad will be spent on the foreign node. Once the pod is running, the *cloudify* script will launch *vncviewer* and the GUI will be available. The picture below shows what has been described above.

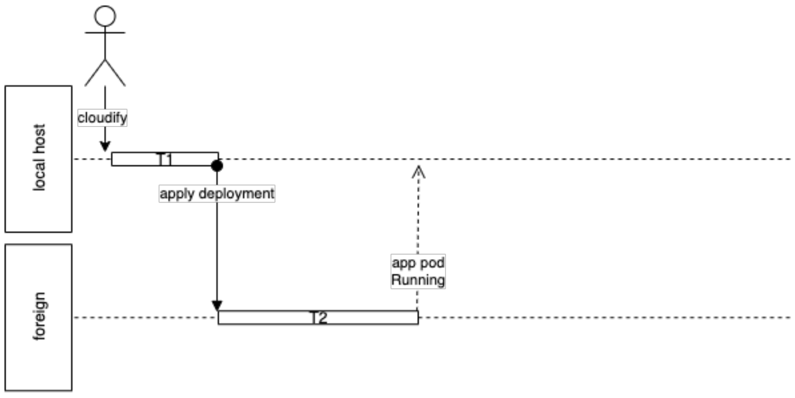

Figure 10: native run mode startup time analysis

The times specified in the picture above have been measured and reported in the table below.

Measure nr.	T1	T2	Tot.
1	1,497 s	4,716 s	6,213 s
2	2,835 s	4,649 s	7,484 s
3	1,909 s	4,865 s	6,774 s
	2,080 s	4,743 s	6,823 s

Note that the last line contains the average values among the three measures for each resource.

As it can be seen in the table above, much of the startup time has been spent on the foreign node.

7.2.2.2. *Firefox execution with client-side Docker container run mode*

This execution mode is very similar to the previous one. The difference is that the client-side is a container instead of native applications. For this reason, the same conclusions discussed in the previous section can be reached.

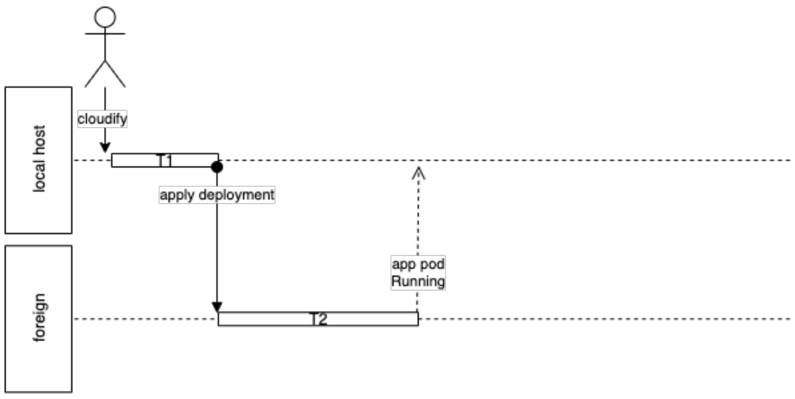

Figure 11: docker run mode startup time analysis

The times specified in the picture above have been measured and reported in the table below.

Measure nr.	T1	T2	Tot.
1	0,764 s	7,759 s	8,523 s
2	2,597 s	4,748 s	7,345 s
3	1,390 s	5,720 s	7,110 s
	1,584 s	6,076 s	7,659 s

Note that the last line contains the average values among the three measures for each resource.

7.2.2.3. Firefox execution with client-side Kubernetes pod run mode

In this execution mode the timeline is a little different. In fact, since the client-side is a pod too, once the target application pod is running, the *cloudify* script prepares the *vncviewer.yaml* file and then to apply the *job*. So, the first part of the execution, it means from the very beginning to the target application pod *Running* status, is the same shown in the previous sections. The time between the target application pod *Running* status and the execution and the *kubectl apply* command for the *job* has been named T3 and will be spent on the local host. Then, the *cloudify* script will wait for the vncviewer pod to be in *Running* status. The time between the *kubectl apply* command and the pod *Running* status has been named T4 ad will be spent on the foreign node.

The picture below shows what has been described above.

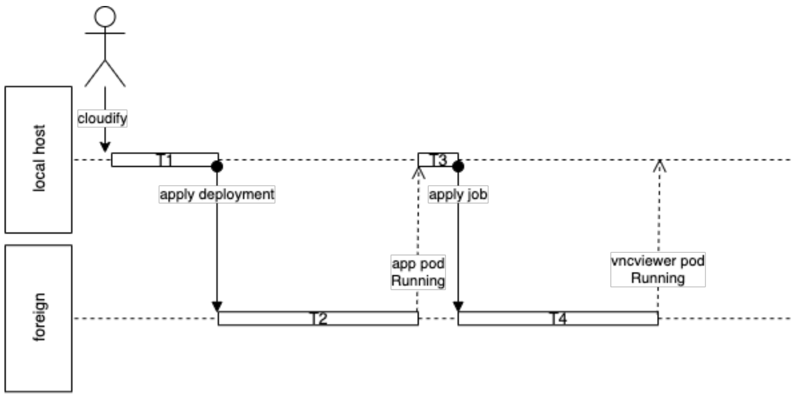

Figure 12: pod run mode startup time analysis

The times specified in the picture above have been measured and reported in the table below.

Measure nr.	T1	T2	T3	T4	Tot.
1	2,256 s	5,645 s	0,105 s	6,736 s	14,742 s
2	1,185 s	5,278 s	0,081 s	6,525 s	13,069 s

3	1,398 s	6,916 s	0,101 s	6,762 s	15,177 s
	1,613 s	5,946 s	0,096 s	6,674 s	14,329 s

Note that the last line contains the average values among the three measures for each resource.

The startup time is much higher than the target native application one, just like the previously discussed client-side execution modes. Obviously, this depends on the fact that this time there are two pods to be scheduled by the Kubernetes cluster. However, considering the benefits given by the fully cloudified execution discussed in the previous chapters in terms of pod reachability and other elements, this might be an acceptable overhead.

7.3. KubernetesOnDesktop on Liqo

So far, the validation on a vanilla Kubernetes installation has been shown. Actually, there is an «open source project named Liqo started at Politecnico of Turin that allows Kubernetes to seamlessly and securely share resources and services, enabling to run your tasks on any other cluster available nearby. Thanks to the support for K3s, which is a lightweight official Kubernetes distribution, also single machines can join a Liqo domain, creating dynamic, opportunistic data centers that include also commodity desktop computers and laptops as well»[52].

The Liqo project allows to virtualize a peered remote cluster as a single node of the user's cluster. It means that the pod scheduling will be managed from Kubernetes as usual and then the KubernetesOnDesktop project may be executed in this context as well. The main advantage of using KubernetesOnDesktop with Liqo is the possibility to use the user's desktop PC or laptop as a single K3s cluster node and offload the desktop application execution on a peered remote cluster. This way, it is possible to exploit all the fully cloudified implementation features even in a commonly used desktop PC or laptop.

For this purpose, the KubernetesOnDesktop project has been tested successfully by creating two VirtualBox virtual machine provided with a K3s single node cluster for each and with the Liqo installation. Once the two clusters peered each other, so that a new virtual node representing the remote cluster has been created on each by Liqo itself, the *cloudify* script has been launched on one of them. Accordingly with the pod scheduling described in the previous chapters, the client-side pod has been scheduled on the local node (it means on the local cluster) and the server-side pod has been scheduled on the other node (it means on the foreign cluster, since it is represented as a virtual node in the local cluster). This way it has been possible to effectively exploit the remote machine hardware capabilities by using the fully cloudified implementation.

[52] Liqo official website – What is Liqo, https://liqo.io/, October 13th, 2020.

Chapter 8 - Conclusions

In this thesis it has been explained how to offload a desktop application execution to a remote Kubernetes cluster node by maintaining its control on the local host through the Graphical User Interface and the audio stream remotization.

As described in the previous chapters, the main architecture has been split in a server-side and a client-side. The first one is in charge of executing the target application and of remotizing its Graphical User Interface and the sound to the local host; the client-side is in charge of showing the remote application's Graphical User Interface and reproducing the application's sound in the local machine. To achieve these goals, it has been used in the server-side a VNC server and a PulseAudio client in order to transmit the audio/video streams, and in the client-side a VNC client and a PulseAudio server to receive those. Moreover, to make the connections secure the SSH protocol has been used in both sides.

To implement the server-side, the containerization technologies have been exploited. In fact, by using Docker it is possible to containerize the target application, along with its dependencies, the VNC client, the PulseAudio server and the SSH tools. This way several issues have been solved, for example, those related to the huge amount of software installations on the remote machine and the lack of control of the resources assigned by the remote host to the target application. In addition, since the remote machine may be a cluster node, Kubernetes has been used as orchestrator in order to avoid a by hand scheduling. For this reason, a Deployment has been created. Once applied on the cluster, it will create in turn a pod executing the aforementioned server-side container. Thanks to Kubernetes the pod will be automatically scheduled on a remote node that has sufficient resources.

The client-side implementation has been realized with three execution modes: native, Docker container and Kubernetes pod. Each one shares an issue related to the remote pod reachability. In fact, since the server-side runs on a remote cluster connecting the client-side to the remote pod is necessary. In order to solve that problem a service has been created on the cluster.

In the native implementation the client-side software has been installed natively on the local machine. This solution presents some issues since the client-side tools can have some execution or configuration conflicts with the pre-existent applications. Moreover, a huge amount of software installation is required, and this will negatively impact on the user experience. In order to solve these issues, a containerized implementation of the client-side has been provided as well. In this case there was a new challenge related to the container audio/video binding with the host's interfaces. This has been respectively solved for the video by exposing the host's DISPLAY environment variable (containing the video Unix socket) and by mounting the host's */tmp/.x11-unix* inode to the container, for the audio by mounting the host's sound card as a drive in the container. In both the previous implementation, the remote pod reachability has been managed through a *NodePort* service. Finally, since the client-side could be a Kubernetes node as well, a Kubernetes Job has been created that, once applied on the cluster, will create in turn a pod executing the aforementioned client-side container. In this case it is possible to exploit the Kubernetes DNS Service feature to make the remote pod reachable from the client-side instead of use a *NodePort* service. However, a problem related to the pod scheduling still exists. In fact, the server-side pod must be scheduled on a remote node and the client-side pod must be scheduled on the local node. For this purpose, the pod Kubernetes *Affinity* feature has been exploited.

Moreover, Kubernetes Persistent Volume Claims have been used to enable the target application data persistency, in order to improve the user experience.

The project tests demonstrate that the desktop application offloading can work without affecting negatively the resource consumption and the user experience. The only sensitive overhead has been recorded for the network traffic due to the VNC and PulseAudio streams. Nevertheless, considering the advantage due to the remote host hardware capabilities exploitation and substantial saving of resources on the local machine, this is an acceptable overhead. Moreover, it is possible to change the VNC video quality and compression parameters in order to obtain a tradeoff between the user experience and the resources consumption.

In conclusion, several features can be additionally implemented in the project, for example a sort of “live cloudification”. It means that once the user launches an application, the system call may be intercepted from kernel module so that the operating system itself can automatically check whether the application can be executed as a remote pod or natively on the host, depending on the network bandwidth availability. This way, the user will not have to take care of choosing what is the best way of executing the target application.

Another additional feature that can be considered is the “hardware remotization”, that is to say a way to control a remote physical peripheral (for instance a robot connected to the remote host through a USB port).

Finally, regarding the *cloudify* script, it may be adapted in order to be used to create a Kubernetes operator. This way the application launch and control will be decoupled from the local machine, since the *cloudify* script itself will be cloudified.

Bibliography

Books:

- Turnbull J., *The Docker Book. Containerization is the new virtualization*, 2019.
- Kane S. P., Matthias K., *Docker: Up & Running. Shipping reliable containers in production*, O'Reilly, 2018, second edition.
- Sayfan G., *Mastering Kubernetes. Large scale container deployment and management*, Packt Publishing, 2017.
- Burns B., Beda J., Hightower K., *Kubernetes Up & Running. Dive into the Future of Infrastructure*, O'Reilly, 2019, second edition.

Websites:

- Docker official website, https://docs.docker.com/, October 13th, 2020.
- Kubernetes official website – Kubernetes Documentation, https://kubernetes.io/docs/home/, October 13th, 2020.
- Vmware official website - Virtual Desktop, https://www.vmware.com/topics/glossary/content/virtual-desktops, October 10th, 2020.
- Citrix official website - What is Desktop as a Service (DaaS)?, https://www.citrix.com/en-gb/glossary/what-is-desktop-as-a-service-daas.html, October 10th, 2020.
- Microsoft official website - Use apps from your Android device on your PC, https://support.microsoft.com/en-us/help/4577326/use-apps-from-your-android-device-on-your-pc, October 10th, 2020.
- OpenSSH official website, https://www.openssh.com, September 9th, 2020.
- TigerVNC official website, https://tigervnc.org, September 9th, 2020.
- Tristan Richardson - RealVNC Ltd. and others, Xvnc − the X VNC server, https://tigervnc.org/doc/Xvnc.html, September 10th, 2020.
- noVNC official website, https://novnc.com/info.html, September 9th, 2020.
- PulseAudio official website, https://www.freedesktop.org/wiki/Software/PulseAudio/, September 9th, 2020.
- Websockify official GitHub repository, https://github.com/novnc/websockify, September 9th, 2020.
- x(7) - Linux man page, https://linux.die.net/man/7/x, September 9th, 2020.
- Rook official website, https://rook.io/docs/rook/v1.4/, September 13th, 2020.
- Peter Leung, How to disable SSH host key checking, https://linuxcommando.blogspot.com/2008/10/how-to-disable-ssh-host-key-checking.html, September 15th, 2020.
- ssh(1) - Linux man page, https://linux.die.net/man/1/ssh, September 15th, 2020.
- Ubuntu manpages, http://manpages.ubuntu.com/manpages/trusty/man1/pactl.1.html, September 16th, 2020.
- Blender official website - About, https://www.blender.org/, October 12th, 2020.

- Nvidia developer official website - NVIDIA Container Runtime, https://developer.nvidia.com/nvidia-container-runtime, October 12th, 2020.
- Liqo official website – What is Liqo, https://liqo.io/, October 13th, 2020.