POLITECNICO DI TORINO

Master's Degree in INGEGNERIA INFORMATICA (COMPUTER ENGINEERING)

Master's Degree Thesis

Mass Scale Lightweight Remote Desktop Environments for Educational Purposes

Supervisors	Candidate
Prof. Fulvio RISSO	Federico CUCINELLA
M.Eng. Marco IORIO	

December 2021

Summary

CrownLabs was born in 2020 during the pandemic as a service for students, to let them practice with laboratory experiences thanks to remote desktop environments. CrownLabs has been started by a group of students and volunteers from Politecnico di Torino, and soon became a real-life playground for experimenting with cutting-edge and production-grade cloud technologies and development techniques (Kubernetes, CI/CD, open-source contributions).

This thesis work aims to extend the CrownLabs ecosystem to make it suitable for a larger number of users, and especially to enable it as an exams platform. Hence, allowing students to interact with the same environments they have been studying onto, but in an insulated and protected sandbox that prevents distractions and cheating.

The work spans from the whole CrownLabs infrastructural design and implementation to the choices undertaken to be integrated with Moodle, the platform that our university uses for delivering computerized exams.

Specifically, the first area of investigation concerned the shift from virtual machines to container-based environments. This remarkably increased the performance in terms of startup time and resources consumption: a single physical machine can host several dozens of remote environments.

Subsequent design and implementation of automatising modules enabled the usage in the exams scenario. Such modules provide: automatic startup of the remote environments prior to exams; pre-population of the environments with custom content; automatic termination of the environments when quiz attempts close; seamless delivery of the content produced inside the environment.

Initial tests over startup times and resources consumption proven the potential of this solution and lead to the decision to leverage CrownLabs during a Computer Science exam session. Through the work done within this thesis, CrownLabs has been adapted for delivering remote environments for the computer sciences exams session held in September: almost eight hundred examinees used PyCharm through their web browser during the exam. The platform has been appreciated both by students and professors of the course.

Table of Contents

Chapter 1

Introduction

1.1 Goal

This thesis has been developed at the Polytechnic University of Turin and consists in the extension of **CrownLabs**, a project which has been developed during the *COVID-19* pandemic started in 2020 by Ph.D. students and master's degree students of Computer Networks and Cloud Computing courses. The initial aim of the project was to enable students of the networking courses to remotely make didactic laboratory experiences but kept on growing and generalizing, not being just a mere service for such courses, but also an actual real case playground project for those who wanted to take part of such a wholesome and enriching experience.

The main purpose of this thesis is to demonstrate how such kind of infrastructure can be extended in order to scale up to bigger numbers, improving the reliability and the overall architecture in order to deliver remote environments to be used for examinations.

1.2 Structure of this thesis

The initial part of this thesis contextualizes technologies and background aspects which are needed to better understand the reasons of this work.

The following design section summarizes how the general infrastructure has

been set up and why.

The implementation section consists in the operative steps that have been taken in order to achieve the main goals.

The validation section is a discussion of the results, along with metrics collected during different kinds of tests that have been run during the various phases.

The last conclusion part illustrates the final thoughts and suggests further steps that could be done to improve and further expand this kind of project.

Chapter 2

Background

As mentioned, this thesis mainly consists in an extension of the CrownLabs project. It basically consists in a graphical interface that lets students and professors to start remote environments (in practice, remote desktops) which contain all the needed tools to accomplish such experiences, with the possibility to share the screen and the control of those environments between the users.

It can also enable the creation of persistent virtual machines that can be accessed at any time that are very convenient for instance to hold development environments without having to pollute the user personal computer.

2.1 Related works

The following sections will give an overview of several products that could be related to CrownLabs and other aspects of this thesis work. In general all of them provide some kind of remote experience.

2.1.1 VLAIB

Polytechnic of Turin technical department developed this technology on top of an on-premise cluster equipped with VMWare Horizon. Its purpose is to provide the same environment present in the terminals present in a LAIB (the university base computer science laboratories): remote desktops delivered by this system

provide the same large package of appliances present in physical LAIBs, which students can use remotely through a web browser. This technology can also be used to solve issues relatively to licenses, compatibility or performance that students might encounter.

The main purpose for such technology has been found in the creation of exams environment: the Polytechnic exam infrastructure[1]) has been integrated with the VLAIB system in order in order to let students interact with a controlled and hardened desktop environment providing the necessary appliances to take practical exams types. Given the complexity of the infrastructure and the large image for the VLAIB virtual machines, such approach resulted to be not easily scalable, thus not suitable for exams with large numbers of students (above the hundred of sessions).

2.1.2 Coderunner

CodeRunner is a free, open-source Moodle module which enables the creation of questions which to be answered by inserting code segments. These can be consequently ran to let students check their correctness. This technology has been tested in the context of the pandemic to renew the computer sciences exams of the Polytechnic of Turin which always have been written on paper mainly due to time and space limits relatively to the LAIBs.

CodeRunner uses a dedicated backend (called Jobe) to run the inserted code in a sandboxed environment. Virtually any language is supported and advanced editing functions may be enabled (such as code completion and highlighting) in the most recent versions of CodeRunner. However, due to the current deployment of Moodle at PoliTo - which has been heavily customized in order to better integrate with the university web services - latest versions of CodeRunner are not compatible and thus several improvements for this plugin (such as code completion and syntax highlighting) are not available. This is one of the limitations which lead to the use of CrownLabs for deploying remote environments in which students could use a real IDE to take their exams.

[1]See Section 2.7

2.1.3 Commercial products (AWS/Azure)

There are several commercial products available from the current leading cloud providers which offer personal environments in the cloud that can be accessed remotely. These kinds of technologies aim to provide a way for the user not to stick to physical devices they own so that any of them, thanks to an internet connection, can be used to access a single remote workspace whenever then need.

The user can generally interact with mouse and keyboard through a web browser or a dedicated application, sending input and receiving the virtual desktop of the remote machine. In some cases also audio input/output could be supported. Often support of other kind peripherals is not granted, mostly because of limitation of the remote desktop protocols used.

This kind of approaches can simplify the management and provisioning of on-premises machines, decoupling the user needs and the machine that has to be assigned to such user.

AWS Workspaces

Amazon Workspaces (from Amazon Web Services) enables the user to create and connect to remote desktop virtual machines through a client which can be installed on the physical machine that is being used to connect from. This provides a rather responsive user experience and easy usage.

Azure Virtual Desktop

Azure Virtual Desktop (from Microsoft Azure) provides a comparable solution, enabling connection through a dedicated client or a web browser.

2.2 VMs based CrownLabs

CrownLabs was born initially to run remote environments based on virtual machines. Most of the times these are ephemeral[2] instances: any file created

[2]The storage provided for these environments is not persistent: they cannot be shut-down or stopped. In case of power outage or crash, data loss may occur.

inside of the remote environment that the user wants to keep has to be copied in a shared folder that is accessible from the remote environment and can be managed also from "outside" the virtual environment.

A full-fledged operative system runs inside the virtual machines, along with a remote desktop software suite (see Section 2.5 for further details) that enables connecting to the instance through a web browser. Most of the virtual machine images enable the user to get root access (over the VM itself), thus install additional software and take full advantage of the remote environment.

2.3 Virtual Machines VS Containers

The classic virtual machine based approach has certain usability advantages and provides great flexibility especially for personal usage, however it also present several drawbacks, especially relatively to performance and scalability.

One of the major issues is given by the overhead that a virtual machine needs in order to be functional. The CPU and memory resources involved can become considerable when the infrastructure needs to be scaled up.

Another issue is related to the resources which generally need to be allocated and reserved wholly to the virtual machine and cannot be shared among other instances.

These issues can be almost completely solved by switching to containers. A container mainly consist in process insulation techniques that enable software to run on a physical machine but in sandboxed environments whose access to system resources is limited to well defined sets. In particular, storage, networking and memory access are restricted to reserved areas which cannot be (easily) circumvented by the insulated process or external ones.

There is no resources overhead required by a supervisor or other system process which are not effectively needed for the purpose of a well defined remote environment.

2.4 Infrastructure of a desktop

This section is a brief discussion of what a graphical desktop is "made of". The heart of a graphical user interface is a *framebuffer*: it can be represented as the matrix of pixels (basically a matrix of colors, one per pixel, with each color being generally an array of three bytes, one per channel, usually red, green and blue) which are being displayed on a screen.

The framebuffer can be either physical - in this case it is directly read from the display adapter which translates the information for a screen to display the picture - or virtual, which means there is no actual display output. This is the case of CrownLabs and generally all the remote desktop services which have been presented. A single physical server would be able to host dozens (when not hundreds) of virtual environments; it would be rather unpractical to have a corresponding number of physical video outputs connected to a single server. The virtual framebuffer instead enables a single machine to host an indefinite number of independent environments, each of them being remotely controllable through a remote desktop protocol.

2.5 Remote Desktop Protocols

One of the core components of what has been discussed up to now, which is in common amongst any of those technologies, is the protocol which enables the actual remote control. There are several protocols and infrastructures which can be used, whether they are open source or proprietary.

Remote desktop protocols enable viewing (usually inside a window of the client device) and controlling (by interacting with such window) a remote desktop, be it a physical or virtual one. This generally works thanks to a bi-directional, asymmetric communication: the client receives the screen content of a remote machine and sends the commands (mostly mouse and keyboard input) to the remote environment.

Such protocols generally let the possibility to set parameters for the connection, such as compression and quality levels, which result in different usage of bandwidth and compute resources, in order to achieve the best trade off between user experience and resources costs.

2.6 Kubernetes

This (rather new) technology plays an important role in the infrastructure from the beginnings of CrownLabs. It has been initially developed by Google and its purpose is to provide a way to manage a cluster[3] and enabling it to host cloud native software.

Kubernetes practically consists in an orchestrator for containers[4]: through specially crafted configuration files it is possible to declaratively define the status of the cluster in terms of running applications, network connection, exposition of services, security and other aspects of the cluster; Kubernetes will keep such configuration as a reference in order to make the cluster state reflect it.

2.6.1 Kubernetes resources

Each aspect of a Kubernetes cluster is defined by a *resource* of some *kind*. There are several predefined resource kinds, that will be discussed in the following paragraphs.

Node

A Node represents a machine (which could be either physical or virtual) which takes part in the cluster. Through *taints* it is possible to define what a Node can and cannot do (for example, Nodes that take part in the *control-plane* by default will not be used for scheduling workloads).

The actual machines identified by Nodes run the effective components which make Kubernetes work.

Namespace

While some kinds of resources are considered "global" within the cluster (they are said *cluster-wide*, other kinds are *namespaced*. A Namespace represent a

[3] A set of servers which share certain conditions, such as the network

[4] Thanks to additional software like KubeVirt, which is used also in CrownLabs from its beginnings, it is possible to schedule and run also virtual machines instead of just containers.

partition of the cluster, which is insulated by certain means. The Namespace resource itself is clearly cluster-wide.

Namespaced resources might refer, from within their specification, to other resources: cluster-wide resources can generally be accessed without issues, while namespaced resources need to be part of the same namespace of the referring one, in order for the reference to work. Interesting use-cases are when the cluster should be shared by different users or to run different applications (in order to further decrease possible attack surfaces). It is also possible to insulate networking between namespaces.

The following kinds of resources are all namespaced.

Pod

The minimal Kubernetes workload unit is represented by a Pod. A Pod is a way to define and model a desired set (which often consist of a single entry) of containers.

Conventionally (although there is no actual distinction within the Pod specification), when more than a container is present inside a Pod, one of the containers is considered the main one, while the other(s) are called *sidecars*. All the containers inside a Pod share the same network namespace and can possibly mount the same volumes which can be associated to the Pod. A Pod will be entirely scheduled within the same node (i.e. a sidecar will not be run in a different node than the main container).

Each Pod has its own network namespace which is bound to a unique IP address within the cluster, which avoid conflicts with port bindings and possible routing issues. IP addresses associated to Pods are ephemeral and thus should not be used to contact Pods. In order to properly access Pods it is necessary to define a *Service*.

Service

A Service represents a backend (usually made of Pods) which becomes accessible through different techniques (e.g. a unique internal IP address within the cluster or a specific port of any node).

Such backend is referenced by means of label selectors. Labels are key/value metadata that can be associated to any Kubernetes resource and can also be used to refer to a certain group of those. When more Pods share the same set of labels, this can be used as selectors, so that any request to the Service can be forwarded to one of those Pods. There is no need to manually create the different Pods though: the *ReplicaSet* resource can automatize such behavior.

ReplicaSet and Deployment

ReplicaSets purpose is to maintain a stable set of running replicated Pods: this is often used to guarantee the availability of a given application by making sure that the desired number of Pods stays up and running, for example in case a pod is deleted, another one will be created. ReplicaSets work by assigning label selectors to the Pods it manages in order to keep track of them.

Deployments are a higher level management mechanism for ReplicaSets and serve to manage what happens to the ReplicaSet. This ease the management of application scaling (if the number of replicas has to be *scaled* up or down), upgrades (when a new version has to be rolled out), rollbacks (when it is necessary to undo an upgrade).

These resources perform any operation to try to keep the availability of a service: in case of an update, replicas are not updated at once. Each of the Pods running the old version of a deployment gets terminated only when a new replica becomes ready to replace it.

Volume, Persistent Volume and PVC

Filesystem in Kubernetes containers provide ephemeral storage, by default: a restart of the pod will remove any data on such containers, therefore, it is not suitable for applications that require to have a persisted state. Within the specification of a Pod it is possible to define *Volumes*[5]: they provides persistent storage to the pod itself. Volumes can also be used as shared disk space for containers within the pod. Volumes are mounted at specific mount points within the containers, defined inside the pod configuration, and cannot mount

[5] A Volume is not an actual Kubernetes resource. It is part of the Pod specification.

onto other volumes or link to other volumes. The same volume can be mounted at different points in the filesystem tree by different containers.

Volumes can be backed up by different technologies:

- an *emptyDir* is backed by an insulated folder on the physical node which exists for the lifetime of the Pod: it gets removed once the Pod is deleted.
- Network shares (such as NFS or iSCSI) enable using different kinds of existing network attached storage systems as backing store for Pods.
- *Persistent Volumes Claims* (PVCs) instead are native Kubernetes (namespaced) resources which are provided by some underlying technology (such as Ceph or again file sharing network protocols) and exist independently in the cluster. Cluster admins can allocate space (using a Persistent Volume, which is a cluster-wide resource) on a certain mean (defined by a *StorageClass*) and cluster users can *claim* such space by binding a PVC to the PV. Pods within a certain namespace can use PVCs in the same namespace as Volumes.

ConfigMap and Secret

A common application challenge is to decide where to store and manage configuration information, some of which may contain sensitive data. Configuration data can be anything, from individual properties to entire configuration files or JSON/XML documents.

Kubernetes provides two closely related mechanisms to deal with this need: *ConfigMaps* and *Secrets*, both of which allow for configuration changes to be made without requiring to rebuild the whole application. Data stored in ConfigMaps and Secrets is be made available to every Pod to which these objects have been bound to and is only sent to a Node if a Pod on that Node requires it, keeping it in the memory on that Node. Once the Pod that depends on the Secret or ConfigMap is deleted, the in-memory copy of all bound Secrets and ConfigMaps are deleted as well. The data itself is stored inside Kubernetes database.

The main difference between a Secret and a ConfigMap is that the content of the data in a secret is base64 encoded. Recent versions of Kubernetes have

introduced support for encryption to be used as well. Secrets are often used to store data like certificates, passwords, access tokens, pull secrets (credentials to work with image registries) and so on.

Custom Resources Definition

As mentioned, Kubernetes works by means of resources. Different pieces of software, called *operators*, monitor resources and the cluster state in order to make the desired state actual in the cluster. While integrated Kubernetes resources (such as those depicted up to now) are managed by components which are internal to the Kubernetes engine itself, it is possible do specify and install custom kinds of resources by using *CRDs*.

A CRD is a cluster-wide resource itself and defines the format that user-defined resources have to comply to. It is then possible to build software to monitor and actualize custom resources that might require it.

2.6.2 Kubernetes components

The following sections illustrate the most important Kubernetes components and behaviors.

etcd

When deployed with a distrubuted configuration, *etcd* implementation favors consistency over availability in the event of a network partition: this means that in case some of the different *etcd*s instances cannot communicate, the database stays consistent but will not be functional until connection is recovered. This consistency is crucial for correctly scheduling and operating services.

Another interesting feature of *etcd* is its *watch API*: clients can subscribe to events that may occur on entities on the database. The Kubernetes API Server exploit such possibility to monitor the cluster and roll out configuration changes or simply restore any divergences of the state of the cluster back to what was declared. For example, if the desired state of an application has been configured so that it has to have three replicas running on the cluster, it might happen that the actual state is not the desired one (e.g. one of the replicas crashed): such difference is detected and an action will be taken to actualize

the desired state (e.g. starting one more replica). In case the operation is not successful, it will be retried more times, usually by adding a back-off time: delay between retries will be incremented with every failed retry.

API server

The API server is another key component of Kubernetes: it serves the Kubernetes API using JSON over HTTP, providing both an internal and external interface to Kubernetes. This means practically every request done by users or an internal agent (for example, an operator) will have to be done through the API server. It processes and validates REST requests, then updates the state of the API objects in *etcd*. This allows clients to set (and get) the desired state which has been mentioned above.

Scheduler

The scheduler is a pluggable component that selects which node(s) have to be used in order to persist some kind of configuration. Basically, it chooses which nodes workloads have to be distributed in.

Given that each pod has a set of resources (requested and reserved) associated to it, the scheduler tracks actual resources usage on each node to ensure that workload is not scheduled in excess of available resources. Different strategies can be adopted depending on which scheduler is used and how it is configured: apart from user-provided resources constraints, directives can be used to "suggest" or "make sure" how the scheduling should be done. Examples include quality of service policies, the fact that some pods have to be scattered across different nodes (anti-affinity) or concentrated in the least number of nodes (affinity), "proximity" to data (in case there some nodes can have some kind of eased availability to certain types of storage).

Controller manager

The controller manager main goal is to run the *reconciliation loop* for the default Kubernetes resources. Such process is the procedure which drives actual cluster state toward the desired one, communicating with the API server in order to create, update, and delete the resources it manages.

Kubelet

Kubelet is responsible for doing the actual scheduling operations on each node and reporting the status of each operation (together with the status of the node itself) to the API server.

`kubectl`

While the previous components are part of the cluster itself, `kubectl` is the official Kubernetes tool to interact with a Kubernetes cluster. As a command line tool, it practically consists in a REST client which is designed to talk to the API server, mostly for getting, updating and deleting resources.

For instance, as Kubernetes resources (while they are written locally) are generally stored on YAML or JSON files, `kubectl` is particularly useful for *applying* such resources. The *apply* operation basically first checks the existence of a resource with a given name (within a namespace, if namespaced), then the resource is either created (if it does not exist on the cluster) or patched in order to be updated with the contents of the YAML/JSON file.

2.7 PoliTO Exam platform

The Polytechnic University of Turin adopted the Moodle platform as part of its didactic web services. The area on which Moodle is most used is for computerized exams.

2.7.1 Moodle

Moodle is a free and open-source learning management system written in PHP. It can be used for blended learning, distance education, flipped classroom and other e-learning projects in schools, universities, workplaces and other sectors.

It offers various kind of features like courses management. Users can be teachers or students for given courses; courses can hold multimedia materials which can be uploaded by teachers and consulted by students. Teachers can create assignments that students can fulfill by uploading files.

The above features however have always been provided by the internal

university web services in a much more structured, organized and efficient way. The reason why Moodle has been integrated into the university infrastructure instead is for enabling computerized exams.

Moodle, in facts, also includes a powerful quiz module among its features. It enables professors to easily build surveys which can be made of several kinds of questions (like single or multiple choices, free text inputs, file uploads and much more) then have students answer compile the quiz in a suitable environment to have the legal validity for being considered exams.

2.7.2 Computerized exams in presence

Up to before the Covid-19 pandemic, exams in PoliTO have always been done in presence. Such exams have always been delivered through computers LAIBs. Most of the exams were based on Moodle quizzes which started in a controlled, full-screen browser window which prevented students from distractions and cheating. A minor part of the computerized exams required the actual use of applications installed on the laboratories terminals.

Exams on Moodle, comparatively to those which require using actual applications, tend to be preferred by professors for several reasons. During the exam, in case of crash of the physical machine or any other issue which might occur, the exam can always be resumed simply logging in on another machine. Moreover, collecting files produced by students through native applications can be non trivial.

2.7.3 Remote exams

The lockdown which begun with the pandemic and the subsequent prevention measures taken by government and the university required exams to be taken remotely, for the first time.

First of all, this required adopting proctoring tools that could be installed in students' computers in order to avoid cheating and distracting. Subsequently, the Moodle infrastructure needed to be improved to support a larger number of sessions.

Chapter 3

Design

3.1 Operators-based infrastructure

Even if CrownLabs could resemble a rather generic web-service, its infrastructure and backend has not been written by "traditional" means. Nowadays regular web-services are realized with custom backends (that can use frameworks like Spring in Java or Express on NodeJS or Flask for Python) which expose an API defined directly by the team behind the project.

The core and backend of CrownLabs, instead, is a Kubernetes cluster. The user directly interacts with Kubernetes API server itself, the source of truth of the whole system is Kubernetes *etcd* database.

Being Kubernetes a rather general purpose system, however, customization is achieved through components that run inside the cluster. These components, called operators, manage the cluster from within the cluster itself. Figure 3.1 shows a simplified overview of CrownLabs infrastructure and its components.

3.2 Kubernetes backend

The core of CrownLabs consists in several operators. Each of them takes care of several resource types, reacting to the actions that users perform on such resources. Many of these types however are not between the integrated

Kubernetes resources (like the ones depicted in the Background): they are Custom Resources defined by *CRDs*.

The main *entities* involved in the CrownLabs infrastructure in facts, are mapped to Custom Resources and the user can interact with them by sending requests directly to a Kubernetes API server on which CrownLabs has been deployed to.

It is possible, indeed, to use whole CrownLabs backend through the `kubectl` command. Regardless of the used client, users can perform any operation, like the following:

- create Instances by applying the relative resource YAML;
- obtain information on how to access Instances from the state of the created instance once it has been started;
- stop Instances (by changing a property in the Instance specification, if supported by the type of instance);
- delete Instances;
- retrieve and change information about the associated Tenant;
- manage other Tenants, in case the user is a manager for a given Workspace.

3.2.1 CrownLabs resources

The following section illustrates the main custom resources kinds that make the whole infrastructure work.

Instances The Instance represents the actual core of CrownLabs: a running environment. The instance specification references a Template that describes which environment has to be run; the instance also includes a status that shows useful information about the running instance, such as how/where it can be reached. Each instance can possibly have more than a single environment associated to it. Each environment can be either graphical or text-based. In the second case it is generally reachable through ssh.

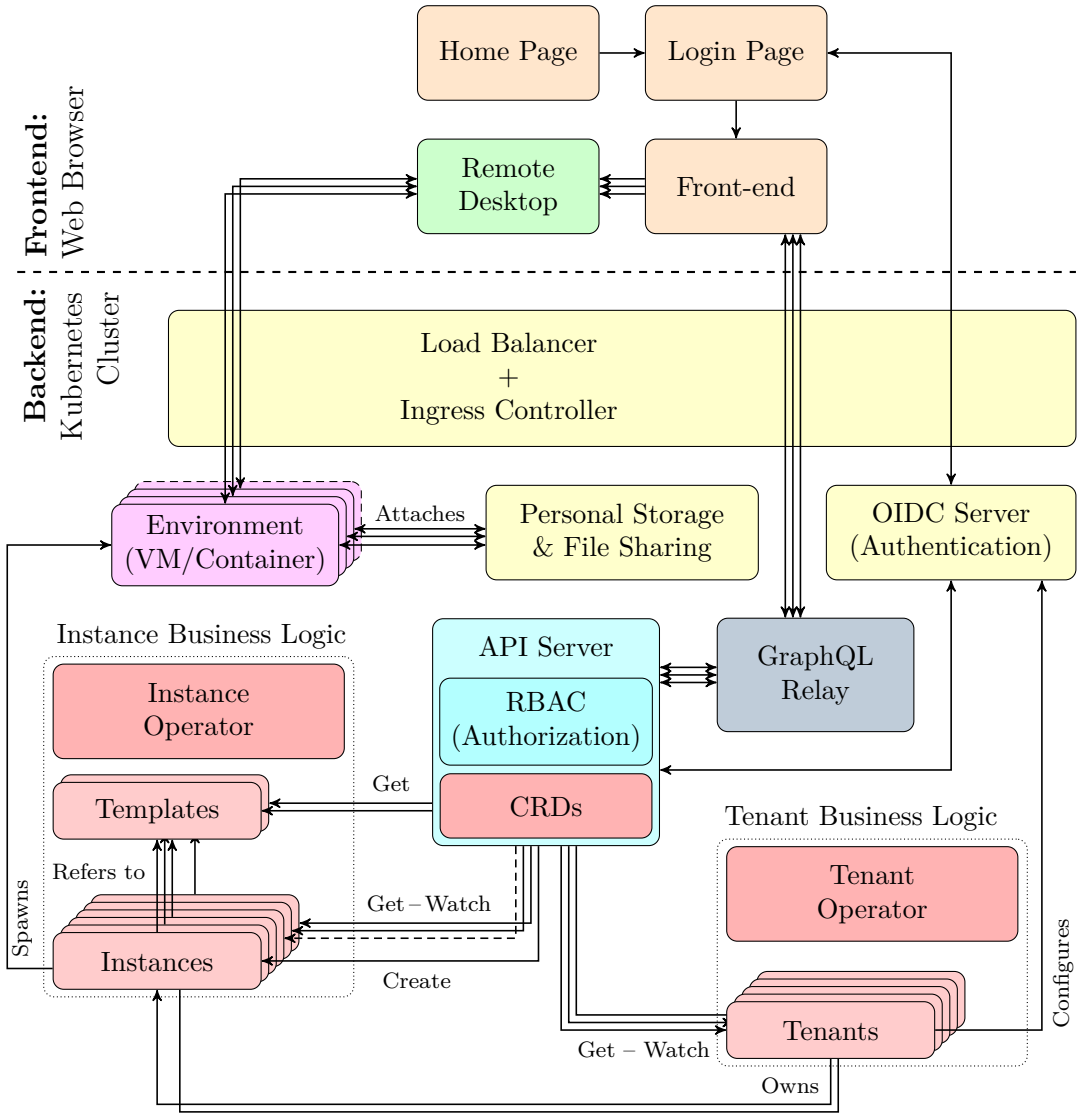

Figure 3.1: Overall schema

Templates A Template can be seen as a "model" that describes how an instance should consist of. Its specification holds information relatively to the image to be used in the instances created from that template, if it the instance will be based on a virtual machine or on a container, the maximum amount of resources that will be available for the environments and other details.

Tenants A Tenant represents a CrownLabs user, whether they are students, professors or administrators. Each user has an associated namespace in Kubernetes, an account on the identity provider system and reserved space on an internal file manager. Thanks to the identity provider association it is also possible to access the Kubernetes cluster using the Tenant credentials.

Workspaces A Workspace often overlaps with the concept of a course. It mostly consists of a collection of Templates. Tenants have one or more Workspaces associated to them, so that they can start Instances based on the Templates that they belong to those Workspaces only. A Tenant can be a User or a Manager for a given Workspace.

3.2.2 GraphQL and qlkube

While `kubectl` is a a great tool for managing clusters while keeping full control over their status, it is not really suitable for users which have no experience with command line application as well as enough knowledge of the Kubernetes infrastructure. There are several graphical user interfaces and dashboards designed for managing Kubernetes clusters, but still they may not be ideal for whose who can not be considered system administrators or still power users, due to the presence of a large number of options and views which could be confusing for the "basic user".

Therefore, since CrownLabs basically consists of a Kubernetes Cluster, virtually any client which supports making REST calls, which are HTTP calls, could be used to be interfaced with a Kubernetes cluster. Specifically, a common web browser could be exploited for running a more user-friendly, possibly custom, interface.

Modern web development techniques however makes use of technologies that ease writing code, both reducing the time needed and enforcing correctness. *GraphQL* is one of these technologies. Initially developed by Facebook, it consists of a language and the relative infrastructure for making special kinds of requests (called *queries* that can be translated into plain REST calls) which include, other than possible parameters needed to accomplish them, which fields are required in the given response.

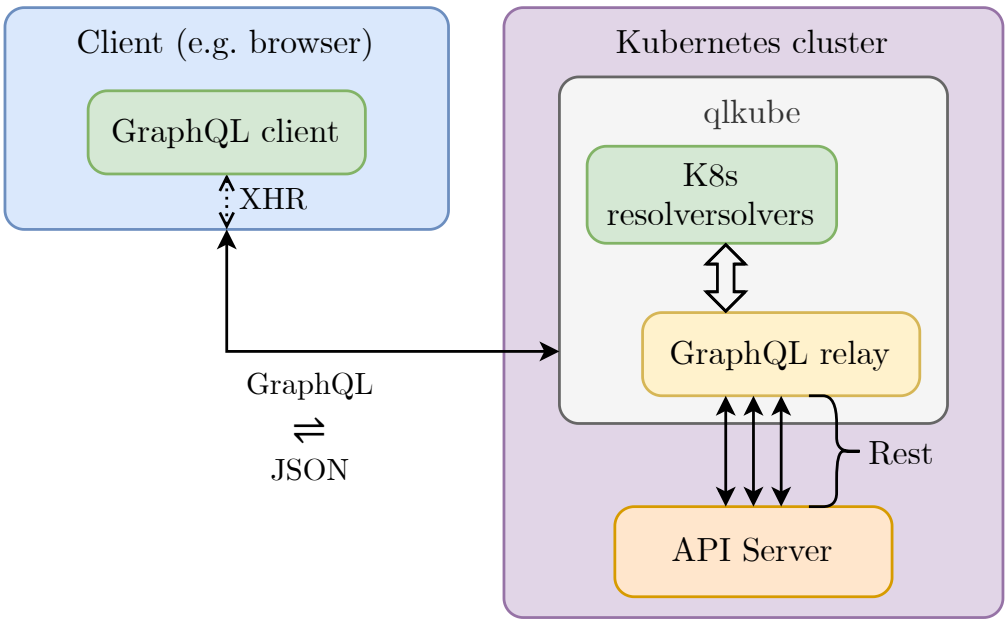

Figure 3.2: GraphQL relay schema

While it is possible to write a backend directly with GraphQL, another way is to have a middleware (like *Apollo Server*) which translates GraphQL queries into regular REST calls. Such kinds of software often offer the possibility of automatically generate the whole GraphQL schema from an existing REST API schema; this is the case of *qlkube*: by using *Apollo Server* as a base, it adapts Kubernetes REST API to be used through GraphQL.

This approach has several advantages over standard REST calls:

- it decouples the data required by the client respectively to the REST interface (for example, a single GraphQL request could be equivalent to multiple REST calls which retrieve an entity and several nested properties)
- it reduces the traffic and the possible elaboration from the backend
- it avoid exposing unnecessary information

Furthermore, Apollo Server also includes a powerful tool (*Apollo GraphQL Playground*) which enables writing queries with the support of code completion and the ability to test the created queries. Ultimately, the counterpart, *Apollo*

Client, is able to generate TypeScript helper functions and types that can be used by a TypeScript based frontend application, which eases development and eases correctness.

Thanks to a modern and consistent backend it is then possible to build a client web application which can be suitable for use by students and teachers without the need of actual training.

Operation

As depicted in Fig. 3.1, the client that is going to communicate with the Kubernetes cluster, can interact with a relay through GraphQL. The Fig. 3.2 focuses on the various steps that take place when a request is made through GraphQl.

The client calls a function on the GraphQL client (e.g. Apollo Client), which prepares the query with the GraphQL language and sends is at a POST request to qlkube. Within qlkube (that runs inside the cluster[1]), the query is processed by the GraphQL server (e.g. Apollo Server) and *resolved* thanks to functions provided by qlkube itself. Resolver function make traditional REST calls to the Kubernetes API Server and are used to compose the final JSON reply that is being sent back to the client.

3.3 Remote desktop management

In case of graphical environments, the user can interact with a remote desktop. Each running instance, in its status, indicates the URL on which its remote desktop is accessible. The web-based frontend automatically shows a button that opens the remote desktop in a new window.

The remote desktop through enables full mouse integration and an almost complete integration with keyboard (some combinations of keys would interfere with both the client operating system and browser and cannot be directly forwarded to the remote environment).

[1]The fact that this component runs inside the cluster is not actually required. In the CrownLabs setup, this choice is made for sake of convenience and simplicity.

3.4 Exams and exercises

Most of this thesis work focused on adapting the CrownLabs infrastructure in order to deliver environments suitable for exams sessions and exams simulations. The following sections illustrate the issues and the design choices undertaken to solve them.

3.4.1 User management

The whole CrownLabs infrastructure is not currently integrated with the Polytechnic Single-Sign-On system and this implies that users of the platform cannot directly log into the platform. Students and professors who are interested in working with CrownLabs have to be registered and managed separately by CrownLabs SSO.

Requiring examinees to register prior the exam is not institutionally comfortable, also considering that a separated login procedure would be required prior to the exam itself.

For the reasons above, combined with the fact that students attending exams should require no access to the CrownLabs frontend dashboard, the exams integration prospects a user-less design. As such, examinees do not have an associated Tenant and are not required/able to log into CrownLabs for taking the exam.

3.4.2 Access flow

In order to access instance, the student who is attempting an exam or a simulation would start from a Moodle quiz (available inside the Polytechnic didactic portal). Since access to CrownLabs has to pass through Moodle, the integration plugin can be the responsible component for authorizing access to instances.

Professors interested in using CrownLabs can insert a special type of question inside the quiz, that will provide access for each student. This way, inside the quiz attempt, each student will find a special link that redirects to their instance.

Authentication

The first iteration prospects no explicit authentication. The exam quiz securely generates a personal access link for each student. Such link includes the following information for each student:

- an user identifier (student's matricola),
- the course identifier,
- an HMAC[2], generated combining the two parameters above, using a fixed key which can be set per each exam session.

The generated link is then used to access the instance. Logging is adopted to check IP addresses which connect to each instance, in order to provide alerts in case of usage of connections from different addresses to the same instance. This approach would still be weak in case students could view and share said generated link.

For this reason the Respondus Lockdown Browser has been used for delivering exams. This proctoring tool indeed provides no way to access and modify the address bar and seeing link destinations, thus the URL of the visited web pages. This partially solves the issue of communicating personal access links to the instances.

As a second step, instances could be authenticated through OpenID connect by the ingress controller. Users must have signed into the university SSO system (access to the quiz implies that the user is already authenticated), which can grants authentication to each instance.

3.4.3 Infrastructure

The exams infrastructure consists by several accessory components that are put beside the existing CrownLabs infrastructure. This section illustrates an overview of such components, which detailed behavior is further discussed in the implementation chapter.

[2]A keyed-hash message authentication code may be used to simultaneously verify both the data integrity and the authenticity of a message. The easiest implementation is a concatenation of the data to be hashed and a key.

Landing

This component mainly acts as an interface between the Kubernetes cluster and Moodle. Links generated by the special Moodle question type point to the Landing component. Given the link, an Instance is identified. The following conditions can occur:

- the Instance exists, it is ready and reachable → the user is redirected to the remote environment desktop;
- the Instance exists but is not yet ready → the user sees a loading spinner until the instance becomes ready, then they are redirected;
- the Instance is not ready → depending on the configuration of the Landing component, the instance can be created on demand or return an error to the user.

Content downloader

This component runs as an init container[3] and is responsible for pre-populating the instance at creation time. This enables customizing each exam session/simulation (possibly also per each student in case it is required) through a field present inside the Template.

Terminator

This component runs for the entire duration of the exam/exercise session. It polls a specific Moodle endpoint which returns which attempts are still open. Once attempts finish, the Terminator component stops the associated instance.

Submitter

This component handles the termination of the instance. Right after termination of each instance, a Job running the submitter component is launched. Such

[3]An *init container* is a special type of container that can be specified in the specification of Pods. It consists in a container which is launched prior than the actual pod containers, which will wait for the completion of any init container specified for the pod.

component collects the content elaborated by the student and uploads it to an endpoint specified in the Template of the Instance.

Collector

This component has been used before the Moodle integration has been completed. It consists in a simple web server onto which files can be uploaded by the Submitter jobs. It also offers a protected file manager graphical interface so that results can be retrieved through an user friendly experience.

3.4.4 Exam lifecycle

The following section illustrates the main actions and actors that would be involved from the beginning to the end of an exam supported by CrownLabs.

Fig. 3.3 shows a simplified schema of the lifecycle of an exemplified exam. The main involved actors include:

- the student, which interacts through a browser;
- Moodle, the exam platform that actually runs the exam;
- the CrownLabs Landing component (see the previous section);
- the CrownLabs Instance that is associated to the student doing the exam.

Exam startup

As soon as the exam starts[4], the Moodle back-end contacts the CrownLabs Landing in order to start the Instance for the Student attempting the exam. The Landing component then creates a new Instance.

Exam fulfillment

The exam runs normally, with the possibility for the student to answer other questions within the quiz.

[4]to minimize, if not nullify, the amount of time that a student has to wait before the Instance is ready, the startup could also be done before the actual beginning of the quiz attempt, on the welcome page for the attempt itself.

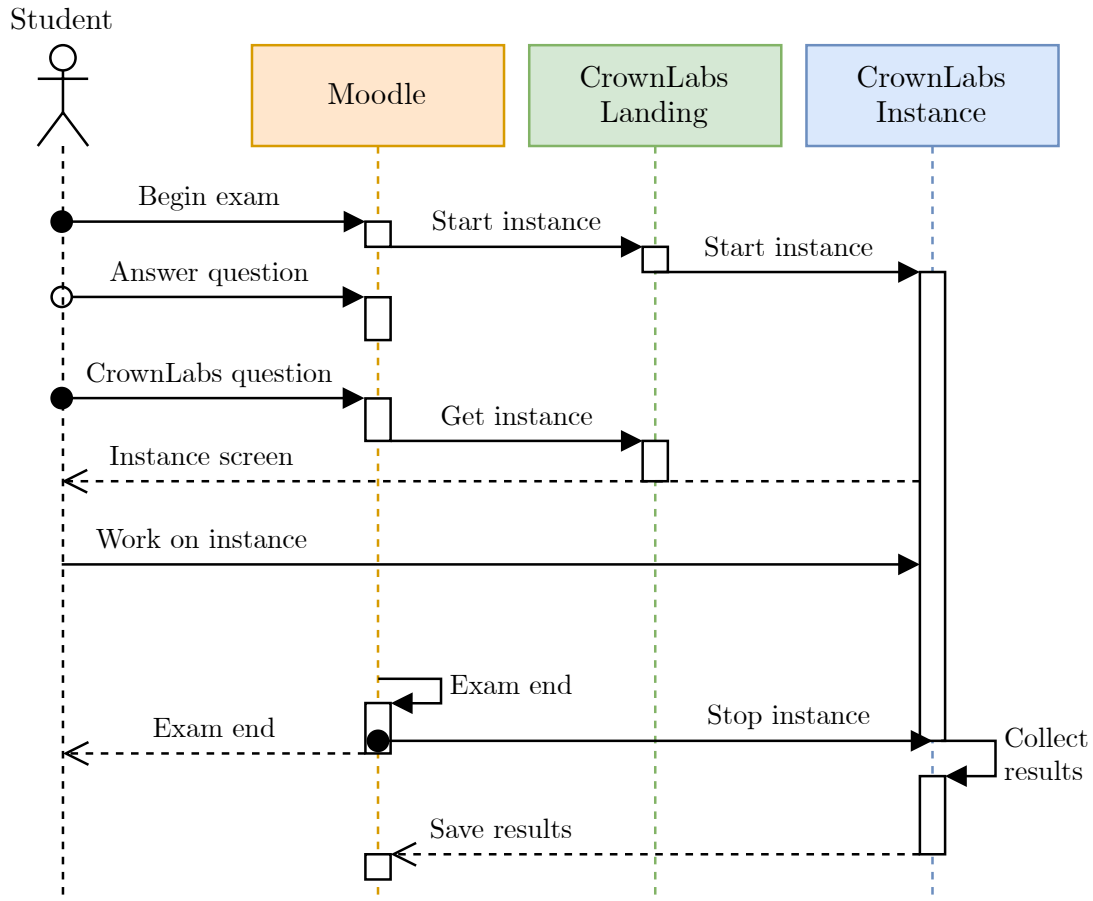

Figure 3.3: Exam lifecycle

A special type of question within the quiz then requests the Instance screen, which is forwarded by the Landing component to the student browser. At this point the attention shifts on the Instance, onto which the student can perform the tasks requested by the exam.

Exam termination

Once the exam ends (in Fig. 3.3 is depicted the case in which the deadline for the exam expires, but the student can also stop the exam on their behalf), the Instance is stopped and results are collected and passed to Moodle.

Chapter 4

Implementation

This chapter illustrates a more detailed overview of the infrastructure implementation relatively to delivering exam environments through the CrownLabs platform.

4.1 Container based instances

It has been assumed that the delivered virtual environments are simple enough to work inside containers. This assumption simplifies the whole solution and eases the management of security. The following sections further elaborate this aspect.

4.1.1 Security context

One of the major aspects that must be considered in this kind of infrastructure is unarguably security.

It is a good security practice to apply the *least privilege* paradigm to containers security configuration: this generally includes disabling root access. This setting, however, prevents basic administrative commands to be executed from within the container, commands that would be required to have an operating-system-like interface. The following exemplary basic features will not be functional.

- Services: proper services management would not be functional. Several system components rely on services. Disabling the possibility to start and manage services and daemon may not provide access to certain kinds of features that might be necessary for some applications to run correctly. In most cases, these include graphical user interface based applications.
- Package managers: system utilities that manage packages (for installing, updating and removing software) practically always require to be run as root. Preventing package management is a major limitation when the aim is to provide a complete desktop environment.
- System configuration: many aspects of the operating system need to be configured through a system account for security reasons. Disabling the root account prevents the user from modifying system configuration files and objects.

A weaker security context which lets the user become root would be required could circumvent some of these limitation, but would expose the whole cluster to high severity security flaws. Even within a container, a process running as the root account can exploit privilege escalation techniques to "get out" of the container and gain access over the physical node. The CrownLabs infrastructure could be potentially used by non-trusted users. For this reason, the security context has to be as restrictive as possible, to avoid the risk of compromising the whole system.

4.1.2 Single application container

Because of security implications and general Docker guidelines, the actual implementation of CrownLabs uses single application instances (for container-based instances). This means that each environment that it would be troublesome to run more than a user application within the same environment. Currently the desktop interface of the container-based instances is stripped down the minimum needed to run the main application. It provides no way to start applications (no launcher menu is available) and the main application is started by the pod itself as it launches. In case the user closes the application it would be launched again by the pod management logic of Kubernetes. The desktop environment is run as another container and has no knowledge of which

applications could be started. Still, there would be no way to start another container within the same pod.

One possibility could be to start a new process from the application container itself: in this case some kind of custom application should be build within the main application container, which could show some kind of window that could then be used to start other applications.

In general however these behaviors are not provided by CrownLabs out of the box and are discouraged.

Container creation

Virtually any application could be made available to be run in a CrownLabs Instance. New applications can be created by just "containerizing" it, i.e., writing a self-contained Dockerfile for the application layer, without introducing strict bindings with the graphical components that would make the deployment monolithic and thus more difficult to maintain. Containers must not include desktop environments, which is provided by the CrownLabs infrastructure itself (see next sections). Two sample applications have been developed for testing purposes but have both been used within laboratories and exams by hundreds of students.

PyCharm The Dockerfile that builds the PyCharm application container starts from a Ubuntu 20.04 image. From this image, several libraries are installed to make PyCharm work. PyCharm CE is then downloaded, extracted and dynamically configured in order to be able to run seamlessly. In fact, before launching the PyCharm executable, a set of pre-configuration files are copied in the proper locations in the resulting image. This allows the software to work out-of-the-box, without any splash screen (e.g., to confirm the license terms), for a better user experience.

4.1.3 Browser based remote desktop: (no)VNC

In order to give users a simple and effective way to access remote environments, several existing protocols can be used. One of the most known protocols that enable remote desktop control is VNC.

VNC

Virtual Network Computing is a graphical desktop-sharing system that can be used to remotely control another computer. It transmits the keyboard and mouse input from one computer to another, relaying the graphical-screen updates, over a network.

VNC is platform-independent: there are clients and servers for many GUI-based operating systems and for Java. Multiple clients may connect to a VNC server at the same time. Popular uses for this technology include remote technical support and accessing files on one's work computer from one's home computer, or vice versa.

The original VNC source code and many modern derivatives are open source and can thus be publicly accessed. It is a robust protocol which is used by many well tested open source implementations.

The core of VNC is the *Remote FrameBuffer* (RFB) protocol, an open simple protocol for remote access to graphical user interfaces. Because it works at the framebuffer level it is applicable to all windowing systems and applications, independently from the platform.

noVNC

The use of VNC however requires a client to be installed on the user's machine. To avoid complicating the user experience and offering a more streamlined access flow, CrownLabs uses *noVNC*: a web implementation of a VNC client. It is both a library written in JavaScript and WebAssembly and a web application that uses such a library. Thanks to noVNC it is possible to access and control instances directly from within the browser. A minimal user interface enables clipboard sharing and management of the VNC connection. Overall user experience, thanks to the WebAssebly implementation of RFB, is highly performant and with a low footprint in terms of network, CPU and memory requests on the client device.

While noVNC surely has many advantages, it surely has one major drawback: VNC protocol works over a TCP and web browsers cannot directly connect to sockets. The solution is to use WebSockets.

WebSocket

WebSocket is a communications protocol that provides full-duplex communication channels over a single TCP connection. WebSocket is designed to work over HTTP ports 443 and 80 as well as to support HTTP proxies and intermediaries, thus making it compatible with HTTP. To achieve compatibility, the WebSocket handshake uses the HTTP Upgrade header to change from the HTTP protocol to the WebSocket protocol.

The WebSocket protocol enables interaction between a web browser (or other client application) and a web server with lower overhead than half-duplex alternatives such as HTTP polling, facilitating real-time data transfer from and to the server. This is made possible by providing a standardized way for the server to send content to the client without being first requested by the client, and allowing messages to be passed back and forth while keeping the connection open. In this way, a two-way ongoing conversation can take place between the client and the server. The communications are usually done over TCP port number 443 (or 80 in the case of unsecured connections), which is beneficial for environments that block non-web Internet connections using a firewall.

Websockify Websockify is a rather small-footprint component that 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, hence allowing a browser (with a suitable JavaScript client) to communicate with any remote server.

This component is required because the client browser needs a bi-directional interaction with the remote desktop, hence using a websocket connection. However, another component is required to translate the websocket connection into standard socket data, to be consumed by the VNC server on the remote server.

The Websockify container has been designed to be executed as a sidecar (see Section 4.1.5) inside the same pod of the VNC server without any further configuration. However, it could be executed in other contexts as well by setting the necessary environment variables, if needed.

In CrownLabs, the `C` version of Websockify has been chosen in order to provide a more efficient (in terms of space, memory and CPU usage) implementation: the "standard" version is written in Python and, as such, implies much more overhead in several aspects.

Compared to the Python version, the C version does not include a static web server, which is used to host the web application that connects to Websockify in order to reach the actual server running in the backend.

4.1.4 Single noVNC deployment

As mentioned, the C version of Websockify is more performing and requires less resources to be run, however it lacks some minor features, for example hosting the noVNC client files. For this reason, the initial implementation of the containerized desktop environment had one more sidecar container per each Pod which served the noVNC client files and proxied the websocket connection to the same http port. The noVNC static files however had no actual need to be present in every instance. This lead to the decision to create a single noVNC deployment (with multiple replicas in order to improve resilience and availability) so that the instances could have made lighter and decoupled the VNC backend from the frontend, for further easing the maintainability.

4.1.5 Sidecar containers infrastructure

In Kubernetes, pods can have more than a single container inside them. All the containers within a pod share the same network namespace[1] and can share volumes and other resources. Conventionally, a pod has a main[2] container while the others, considered support processes, are called *sidecars*. As mentioned, the first implementation included a third sidecar which hosted the noVNC client files.

[1] Containers generally run in insulated network namespace, which means that their network is insulated. Within a pod instead, processes in different containers can directly reach each other as if they were run in the same container.

[2] The fact that one of the containers is the main one is just a convention since it has no actual implications in the configuration of the Pod Specification, nor in the way they work.

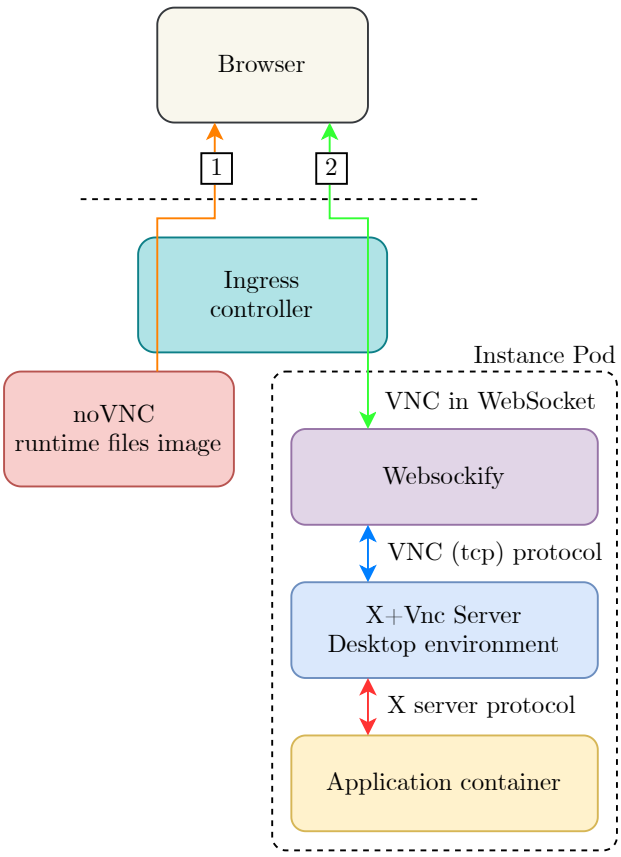

Figure 4.1: Container based instance graphics infrastructure

A container-based instance basically consists in a pod which runs the components described above as sidecars. See the Fig. 4.1 for a clearer representation. The application container runs along with two other containers: one runs Websockify, while the other holds the actual desktop for the instance.

XVnc Server

Linux desktop traditionally works thanks to the *X Window System*, generally with its most famous implementation, *X.Org*. Summing up, it works by managing a framebuffer (practically, a matrix of pixels that can be stored in the system memory or in the GPU memory) and exposing (high and low level) directives to interact with such framebuffer to other applications, usually by means of Unix sockets.

The XVnc server 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 was a normal X display, but they can only be accessed via a VNC viewer. Technically, this consists in two different servers (the actual X server and a VNC server) that are bound together and provide a virtual desktop both to the user (through the VNC protocol) and to the final application (through the X core protocol).

There are several implementations of such type of server. For CrownLabs, TigerVNC has been chosen, since it provides automatic desktop rescaling features. Note that TigerVNC server cannot be considered a regular X server since it cannot display the desktop on a physical screen.

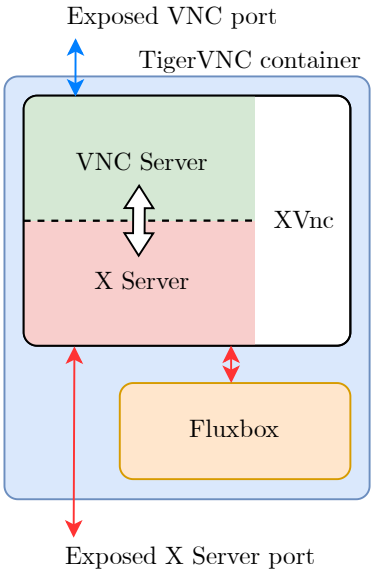

Figure 4.2: X+VNC container structure

This container also includes Fluxbox, a lightweight window manager which provides an effectively desktop-like user experience and, above all, it enables menus and sub-windows to be rendered correctly. The whole container is based on the Ubuntu base image and the X server instances starts with no authentication. This implies that any X client would be able to connect to such server and display content on it, if the IP address of the Pod is known. Network policies can be used to avoid such behaviors in case it is necessary.

In order to make this component work in a containerized environment, it is required to assert the following aspects:

- The `DISPLAY` environment variable must have the same value in the X server instance and in the application container. This requires the component that coordinates the deployment of the two containers to set the `DISPLAY` value accordingly when starting the two Dockers.
- The X server will create a socket file in folder `/tmp/.X11-unix/`, which may need to be shared with the container running the application. This requires the component that coordinates the deployment of the two containers to possibly mount the same shared volume under `/tmp` when starting the two Dockers.

4.2 CrownLabs Infrastructure

This section illustrates implementation details of the CrownLabs infrastructure, with a particular focus on the areas that influenced the use of CrownLabs for exams.

4.2.1 Instance operator

The Instance Operator is the core component of CrownLabs. Like the other CrownLabs operators, it is written in Go language and makes use of the Kubebuilder framework. It simplifies writing Kubernetes software by scaffolding code for controllers and CRDs.

The Instance Operator main task is the Instance Controller. Controllers main method is called *Reconcile*: upon creation, update or deletion of the managed resources, such Method is invoked. Its purpose is to actualize the resource within the cluster. The original Instance Controller of CrownLabs was responsible for starting and managing Virtual Machine Instances. This thesis work added the support for container based instances. The schema on Fig. 4.3 shows a simplified overview of how a container based Instance is actualized on the cluster.

For each Instance the following native Kubernetes objects are created.

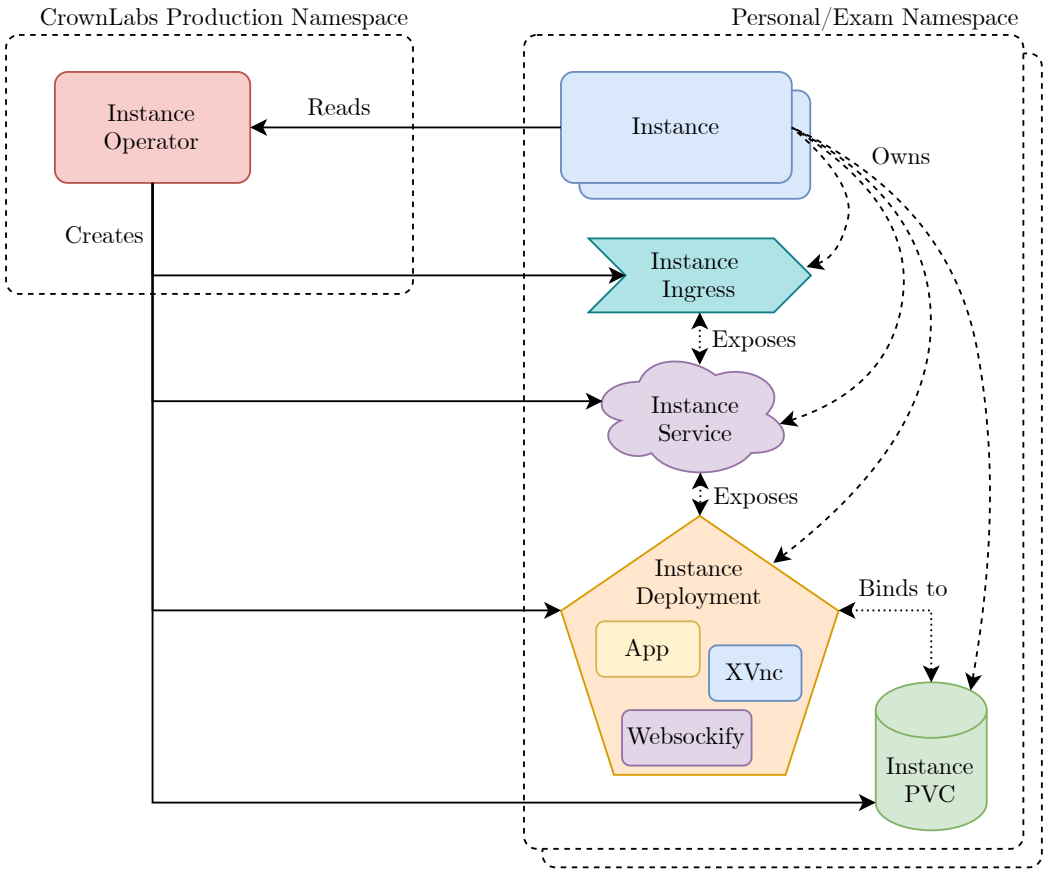

Figure 4.3: Instance Operator

- The Instance Deployment: this is the actual workload that carries the Instance application. The pod created by the deployment includes what is necessary for creating the instance desktop.
- The Instance PVC, which provides persistent storage for the Instance. This is particularly critical during exams, since provides a certain resilience against failures.[3]

[3]Depending on the Template specification, instances can be either persistent or not. For this this work, only persistent Instances are taken into account, since such persistence is crucial during Exams. Persistent storage provided by Kubernetes in CrownLabs cluster is replicated and highly available. In case of failure, contents of persistent Instances is not going to be lost. Also theoretically, in case of over-provisioning of cluster resources, this

- The Instance Service which exposes the Deployment within the cluster.
- The Instance Ingress makes the Instance available over the Internet, through an *http* endpoint that is written inside the Instance status.

All of these resources are set to be owned by the Instance itself. This way, deleting the Instance resource results in deleting all the associated resources without the need for the Instance Operator do to that "manually".

Instance lifecycle

The following section illustrates in detail which components are involved in the Instance lifecycle and the actions they perform.

In case any Instance is created, modified or deleted, since the Instance Operator is *watching* the Instance resource type, it will be notified by the Kubernetes API server. This triggers the `Reconcile` method within the Instance controller. See Fig. 4.4 to follow the main steps within this method.

The first action is to enforce[4] all the required resources, as mentioned in the previous section. When creating the Deployment for the Instance, the operator checks the Instance Specification for the presence of the `SourceArchiveURL` property: in case it is set, the created Deployment will include an *InitContainer* that will run the *ContentDownloader* component (see the Content downloader paragraph for further information).

An Instance can be created as not running: in this case all the resources are prepared but the number of replicas of the Instance Deployment is set to zero. As the Instance is set to be *Running*, the Deployment is scaled up and the Instance is actually started.

At this point, the *Reconcile* method of the Instance Controller stops. The Instance Operator however does not only include the Instance Controller: between the subroutines that are performed by the operator, there is a periodic

enables fault tolerance in case one of the cluster nodes fails: instance deployments can be re-scheduled on other nodes, with a small downtime for the user, without losing the stored data.

[4]this operation is idempotent: if one of the resources already exists, the operator will check that its specification, labels and annotations are correctly set, correcting them if necessary.

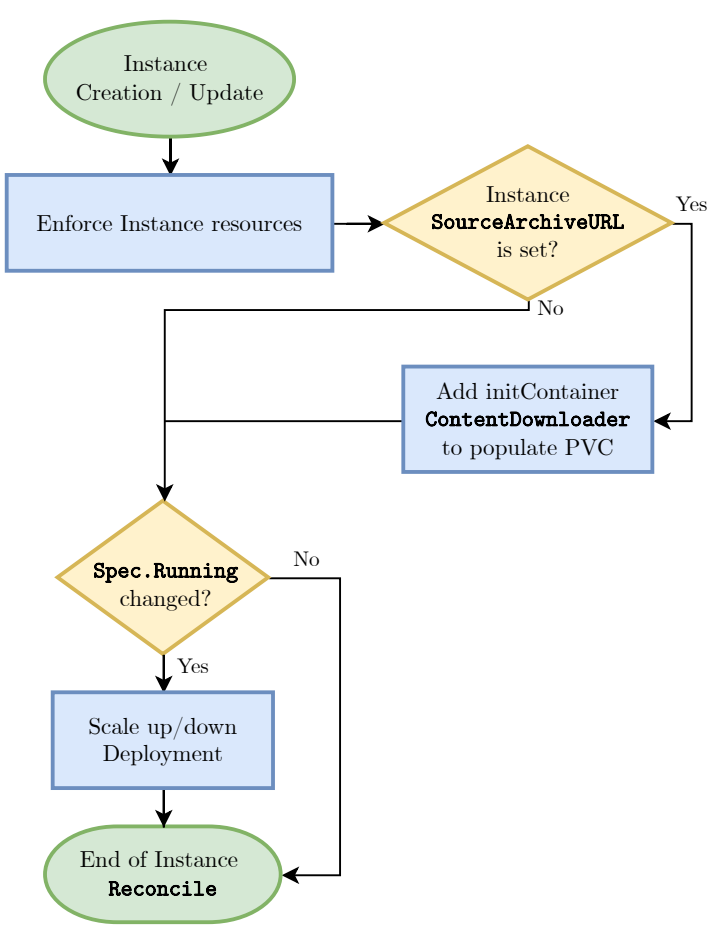

Figure 4.4: Instance lifecycle - first part

check on all those running Instances that include a `StatusCheckURL` field in their specification.

The endpoint provided in such field is contacted - as said, periodically - to check the desired status of the Instance. When said endpoint returns an appropriate value, the associated Instance would be set as not running.

As shown in Fig. 4.5, this change triggers the Reconcile method of the Instance controller[5] that first scales down the Deployment. Then, in case

[5]The Go routine that checks the external endpoint for the Instance desired status cannot directly invoke the Reconcile method. Instead, an update is sent to the Kubernetes API

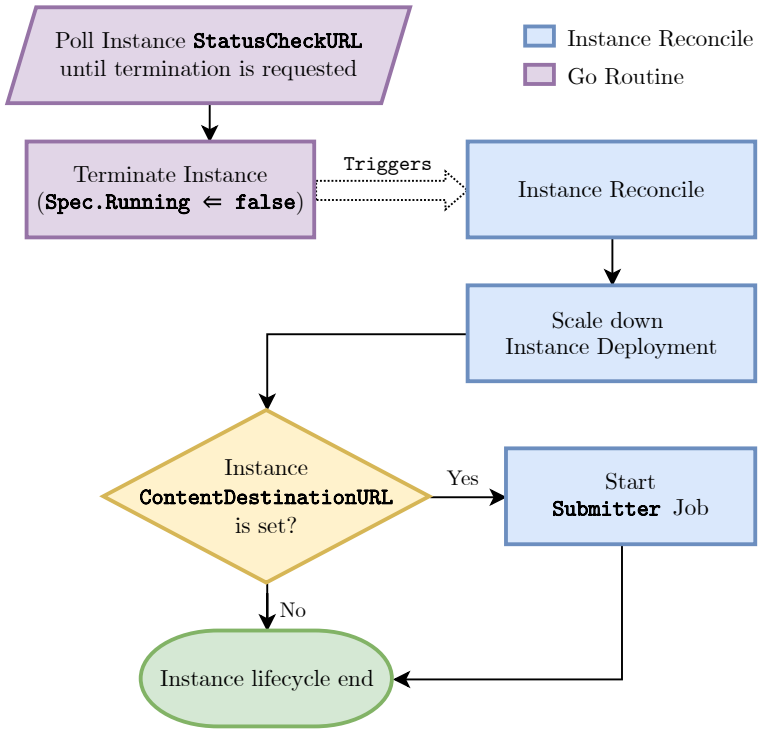

Figure 4.5: Instance lifecycle - last part

the Instance has a `ContentDestinationURL` in its specification, a new Job (running the `Submitter` component) is started to collect the content generated inside the Instance.

4.2.2 Monitoring

Cluster Monitoring is the process of assessing the performance of cluster entities either as individual nodes or as a collection. Cluster Monitoring should be able to provide information about the communication and interoperability between various nodes of the cluster.

CrownLabs monitoring is achieved with the following set of instruments.

Server, which in returns, notifies the Instance Controller, that will run the Reconcile method. The Instance status check routine could in fact be done by an external component, it has been inserted as part of the Instance Operator just for convenience and organization reasons.

Prometheus is an open-source systems monitoring and alerting toolkit. It basically acts as a time series database. Whatever service that should be monitored exposes a `/metrics` endpoint that is *scraped* at regular time intervals. Targets to be monitored are discovered via service discovery or static configuration. Thanks to *PromQL* it is possible to query the Prometheus database and obtain desired data.

Thanos leverages the Prometheus 2.0 storage format to cost-efficiently store historical metric data in any object storage while retaining fast query latencies.

Grafana is an open source visualization and analytics software. It allows to query, visualize (see Fig. 4.6), alert on, and explore metrics gathered by Prometheus.

Figure 4.6: Main Grafana Dashboard in CrownLabs

This monitoring stack has been crucial during the exam session held in September: having a comprehensive visual of the cluster status can prevent issues to happen before it is too late. Resource usage metrics above all, can

be an indication of malfunctions that can be mitigated before they become critical.

4.2.3 Single Sign On

Users in CrownLabs are modeled by the Tenant CRD. The Tenant operator takes care of preparing all the resources associated to every user: a personal Namespace, authorization resources like RoleBindings and ClusterRoleBindings for accessing Templates, and an account on the Identity Provider that has been chosen for Crownlabs, which is Keycloak.

Keycloak provides Single Sign On authentication: when an user needs to log into any CrownLabs service, they would be redirected to Keycloak, that would proceed to authenticate such user for the given service. Further integration can be introduced, for using the university IDP as a source for authenticating with Keycloak. This would enable users to be directly authenticated if they already logged into any web-service provided by the university.

4.3 Exams and Exercises

The integration between CrownLabs and the university web services has been done in different ways. An initial iteration used an API exposed by the didactic portal, which returned information about students registered to courses and exams. A second iteration left this approach to focus more on integrating with Moodle.

4.3.1 PoliTO Exercise platform

The Polytechnic University of Turin offers to its student the possibility to make practice in sight of the exam on a platform that is practically a clone of the one used for exams. For this reason, the initial CrownLabs integration with Moodle has been done through Exercise.

While for exams the number of students is known *a priori* and instances startup can be scheduled before each turn of exams, it would not be possible (with the current resources) to start up an instance per each student registered to every course. For this reason, the Landing component (see Section 3.4.3)

is configured to start instances when they are not found, upon visit of the generated link. This allows starting up instances on demand.

Limits on the maximum numbers of instances is configured, so that an message can be displayed when resources limits on the cluster have been reached. This way students can retry at a later time.

4.3.2 PoliTO portal API interfacing

The integration done for the exercises platform has proven to be valid and functional, however it presented a major drawback. Instances would be started the first time the student tries to open the Instance itself. This would result in a waiting time for the student which can be critical during an exam.

For this reason, the initial integration of the CrownLabs infrastructure with the PoliTo didactic portal initially prospected an external agent. Such component would use the didactic portal API to retrieve the list of students booked for a certain exam and create all the necessary resources for a given exam, then start every instance in advance, right before the beginning of exam. This method has been used for the tests held in September and solved the problem of the waiting time for students.

This solution however has been proved not to be easily manageable and automatable for the following reasons.

- Some logic would have been created for automatically preparing the environment for each exam, upon creation of the Moodle quiz.
- The didactic portal and the exams platform are rather weakly integrated. Students who book an exam are not bound to a single Moodle quiz, the student can choose which quiz to start in case more than one is available. In case different quizzes require different environments in CrownLabs, it would not be feasible to start an instance for each possible quiz and each student who registered for such exam.

4.3.3 Moodle integration

A better integration has been designed in a second iteration. This solution moves part of the logic into the Moodle plugin, removing the integration with

the didactic portal. This way CrownLabs instances are started when the user lands on the quiz attempt page. On that page, the quiz has not been started yet and there is a time window during which the instances have some time to start before the exam begins. When the student navigates to the special question type that shows the link towards the instance, a loading page will be shown in case the instance is not yet ready, automatically redirecting to the desktop when it is ready. This solution reintroduces the issue of the waiting time for the students, but such time would be rather shorter, if not none, since the instance is started upon beginning of the quiz attempt.

This way it is possible to bind a Template directly to a Moodle question. Only effectively used instances are created (so removing the possible overhead of creating instances that will not be used by students that do not show up for the exam) and with the correct environment.

4.3.4 Works delivery

The infrastructure set up for the September section used a rather simple method for collecting material produced by students during the exam. Upon termination of each instance, the Terminator component creates a job (running the Submitter component) that uploads the contents elaborated by students into a rather rudimentary http server. In a second phase, collected results have been manually sent to the respective professors through emails.

The new Moodle integration introduces an endpoint on which uploading such contents, so that professors can download them directly from the quiz review.

4.3.5 Infrastructure

The following section better illustrates the technology behind each component which takes part in the exams environment. Each component is designed to be run inside the CrownLabs cluster as a container. Further details about the implementation depend on the component and are discussed in the next sections.

Landing

The Landing component is written on Go language and includes the Kubernetes client. For this reason it is necessary to provide a service account with the right RoleBindings to the Pod that runs this component, in order to enable it to manage instances.

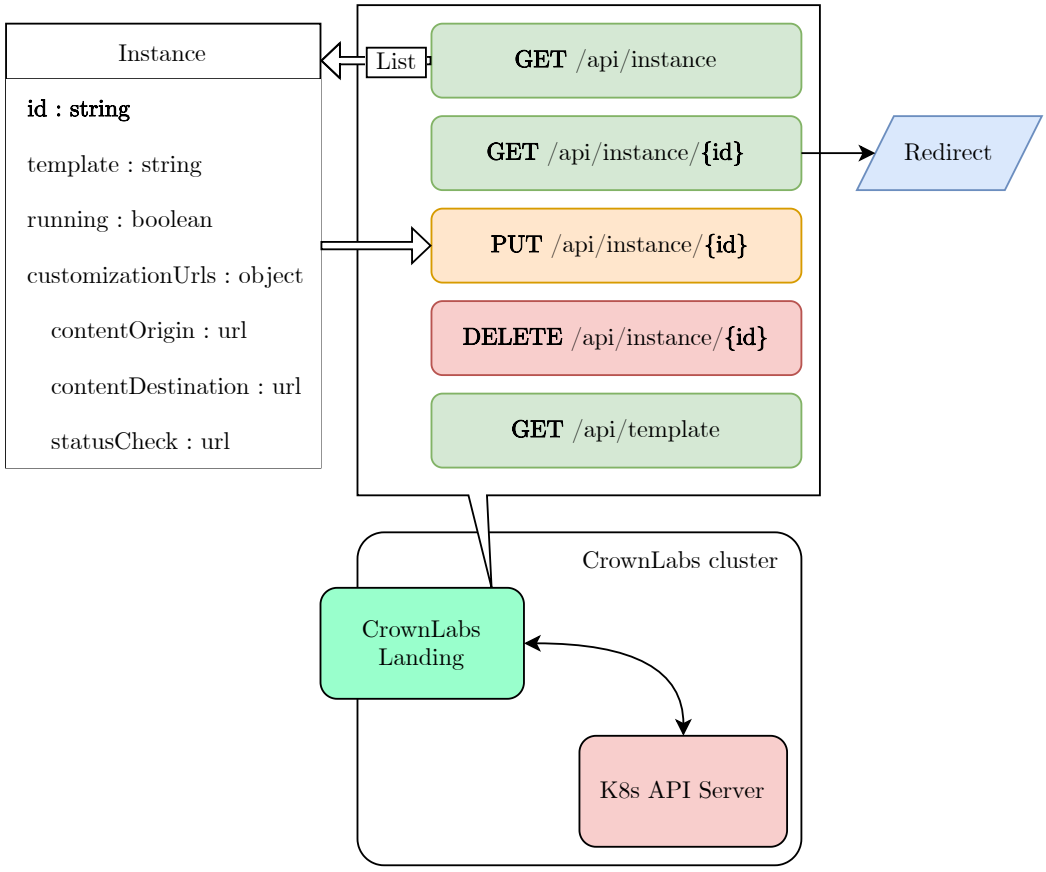

Figure 4.7: Landing component API

As mentioned, upon each request, the logic of this component is able to retrieve Instances from a given namespace and possibly create them, when the startup flags are set to enable such behavior. The webserver contained in the Landing component then replies back to each request with a redirect response or with a user-friendly placeholder that communicates the instance is being created and is not yet available.

Content downloader

The Content downloader is a simple utility script that is built from Ubuntu base image and includes *cURL* and *unar*.

cURL is a command-line tool for getting or sending data including files using URL syntax. It supports HTTPS and performs SSL certificate verification by default when a secure protocol is specified such as HTTPS.

unar is a command-line tool for uncompressing archives. It supports a great number of archive formats.

This container accepts two parameters: the first is the source of an archive to be downloaded. It is passed to cURL, which saves the file to a temporary location. Subsequently, the second parameter contains a destination onto which the downloaded archive has to be extracted.

It is worth mentioning that since, this component is made for being run inside a container, the destination folder has to be specified in the volume mounts at runtime, otherwise the downloaded content would be destroyed upon termination of the container.

Collector

This component is a rather simple webserver written in Go. It has been initially written to accept contents from virtual-machine-based Instances, thus providing a simple web page onto which students could drag-and-drop files to be uploaded. A simple backend accepts POST requests with binary encoded body that is uploaded into a folder specified from the startup arguments.

In case of container-based Instances, the backend is not protected: a Network Policy restricts access to this container only from Submitter Jobs.

In case of virtual-machine-based Instances, a *JWT* is passed as a query parameter to recognize and authenticate the user. Such parameter is saved into the Virtual Machine on startup through CloudInit.

Submitter

The Submitter component is written in *Node.JS*. Its purpose is to process all the files within a folder (passed as an argument to the container at runtime) and submit it somewhere. The initial implementation used for the September session first uploads the single files to the Collector component using *Axios* (a JavaScript library for making http requests) through a `POST` request; subsequently it creates a zip file containing the same files and emails them to an email address that is built at runtime from a string passed from a configuration file and environment variables.

The second implementation creates a zip file and uploads it to an endpoint which is passed as an argument.

In order for this component to work it is clearly necessary to mount a volume to the same directory that is passed as an argument, as the root of the files to be submitted.

Exam-Agent and Terminator

In the first iteration, for the September session, a single component could manage the "lifecycle" of an exam. Written in Go, it includes the Kubernetes client (see the implications of the Landing component) and is interfaced with the university didactic portal. It manages the whole lifecycle for a given exam.

- Prior to the beginning of an exam, it creates all the resources necessary to hold all the resources of the exam:
  - the Namespace for the Exam that contains all the other resources;
  - a NetworkPolicy to avoid unnecessary network operations;
  - Deployment, Service and Ingress for the Landing component;
  - Deployment, Service and Ingress for the Collector component, if necessary.
- If set with an appropriate flag, it creates and launches all the Instances right after creating the exam resources.
- It waits, cycling all running instances and polling onto a given endpoint

that returns if the associated Moodle quiz attempt is still running. Once the attempt is finished, the Agent/Terminator terminates the Instance.

- Upon termination of the Instance, a Job running the Submitter is launched, mounting the same PVC of the Instance, so that what has been produced inside the Instance can be sent to the Collector component.

The second implementation is much simpler, since the Exam-Agent is not required anymore. All instances are being started on demand in a single Namespace which is already present on the cluster. The termination component has been integrated inside the Instance Operator itself. A routine periodically checks the running instances and polls an URL written directly inside the Instance specification: if such endpoint returns a 404 error, the instance would be terminated and a Submitter Job would be created.

4.3.6 Infrastructure hardening

While the standard CrownLabs network is rather permissive also to let student experiment with the infrastructure, exams namespaces are hardened through restrictive network policies. This is achieved thanks to Calico, the CNI used in the CrownLabs cluster, which enables writing precise rules that mostly work through a white-listing model. The only allowed traffic towards the Internet is towards the port exposed by Websockify. Then some service traffic is allowed within the namespace, e.g. communication between the Terminator and Moodle or between the Landing and the Kubernetes API Server.

File-system redundancy: Ceph

CrownLabs cluster is equipped with Ceph: a distributed filesystem without single point of failure. This ensures that the whole system stays healthy even if one of the nodes crashes or physically breaks.

Backup system: Velero

Kubernetes provides a powerful and reliable infrastructure, however the risk of making remarkable damages because of the human error is an actual issue. For this reason, Velero has been deployed on the cluster. It consists of a backup

system which can periodically save every resource in a Kubernetes cluster, also theoretically easing migration and disaster recovery.

Velero is a backup and migration cloud native software designed to perform disaster recovery and migrate resources across cluster. The main purpose of Velero in CrownLabs is to mitigate the following eventualities:

- failed or bugged updates
- unintentional resources deletion
- catastrophical physical cluster destruction/loss of all control plane nodes[6]

Note that storing backups on the cluster itself could help to mitigate unintentional resource deletions but it is not a great strategy, since the deletion of volumes or related resources could lead to the deletion of backups themselves.

Velero can store backups of Kubernetes resources and possibly associated volumes on several cloud native/proprietary storage solutions (such as AWS, GCP, Azure, VMWare, OpenStack, etc.).

In case of full bare-metal environments, it is possible to use Minio, an open source, Kubernetes-native object storage solution which is compliant with Amazon's S3 API. This provides a valid storage target for Velero, provided that the mounted data path is not being stored on the cluster itself.

In the case of CrownLabs, while Minio is deployed on the cluster itself, the storage backend that is used is provided through iSCSI by an external NAS server. See Fig. 5.1 for further information about the cluster configuration.

Infrastructure resilience: multi-master K8s cluster

While the original intent of CrownLabs was to provide a best-effort service, the fact of using it within exams required further work to improve the resilience of the infrastructure. The initial deployment of Kubernetes inside the CrownLabs cluster prospected a single master node, running as a virtual machine within

[6]this would require backups to be stored in a physically different/distant location relatively to the cluster

one of the physical nodes. In case the physical machine running the master node encountered any issue, the whole cluster would fail.

The solution implied adding replicating the virtual master nodes into two more physical servers so that in case of failure of one of them, the platform could keep working minimizing service losses.

This configuration is knows as high availability Kubernetes deployment. Master nodes run several critical services, such as the API server and the *etcd* database[7] and need to be in odd cardinality (in order to reach the quorum in case of inconsistencies).

Network scalability

Kubernetes offers powerful load balancing capabilities that, however, can not be completely enabled because of infrastructural limitations of the place in which the physical cluster has been placed.

MetalLB MetalLB is a load-balancer implementation for bare metal Kubernetes clusters, using standard routing protocols. Among the configurations it offers for providing load balancing capabilities there are *ARP* and *BGP* working methods.

ARP This is the technology that has been initially deployed on CrownLabs, since BGP was not initially available. In this way, the cluster nodes elect one of them as the one that will receive all the traffic through a virtual ip that is announced on the physical network through *gratuitous ARP*. At that point, such node will forward the incoming request to the node that actually holds the required service. This configuration is fault tolerant (since in case of failure of the elected node, another node will be elected, with rather short down-times) but demands load balancing to one of the cluster nodes, thus not being *real* load balancing and possibly becoming a bottleneck.

[7]The *etcd* servers can be actually run on external nodes that are not part of the cluster itself. Deploying etcd directly within the master nodes however is much simpler and further eases management tasks.

BGP This second way requires the routers present on the physical network to support this protocol in order to properly work. In this configuration, thanks to the multi-path routing capabilities of BGP, the campus routers can act as actual load-balancers, scattering requests across the whole cluster. In case the request is being forwarded to a node that does not host the required service, the request is forwarded to the correct one.

Thanks to the network management team of the university though, BGP has been activated on the network segment of the cluster and MetalLB has been configured accordingly, enabling proper load balancing across the cluster.

Chapter 5

Validation

The whole infrastructure has been tested in different ways, first to better understand the feasibility of the solution. Secondhand, results have been collected to understand the effective scalability potential.

After the initial feasibility tests, the CrownLabs infrastructure has been used to carry out a real exam case, within the exams session held in September 2021 for the Computer Sciences courses.

5.1 Testing conditions

CrownLabs runs on a bare-metal Kubernetes cluster made of 6 physical servers with the following specifications:

- 4 Dell PowerEdge R740x servers, each one with
  - 1 Intel Xeon (28 virtual cores),
  - 256 GiB of RAM,
  - 1 TB of SSD storage;
- 2 Dell PowerEdge R740x servers, each one with
  - 2 Intel Xeon (64 virtual cores each),
  - 512 GiB of RAM,

  - 8 TB of SSD storage;

- 1 QNAP TES-1885U offering iSCSI storage (used for backups);
- 1 Cisco SG350X switch providing 1 Gbps interfaces for maintenance purposes;
- 1 Cisco SG350XG switch providing 10 Gbps interfaces for the data plane.

Overall the cluster provides 336 virtual cores, 2 terabytes of RAM and 20 terabytes of SSD storage. Each server is connected with aggregated 10 Gbps links to the data plane switch and with 1 Gbps to the maintenance switch. The two switches are then connected to the Campus network, respectively with a 10 Gbps link and a 1 Gbps link. The overall structure is depicted on Fig. 5.1.

The machines are physically installed inside the university and are managed by the CrownLabs team.

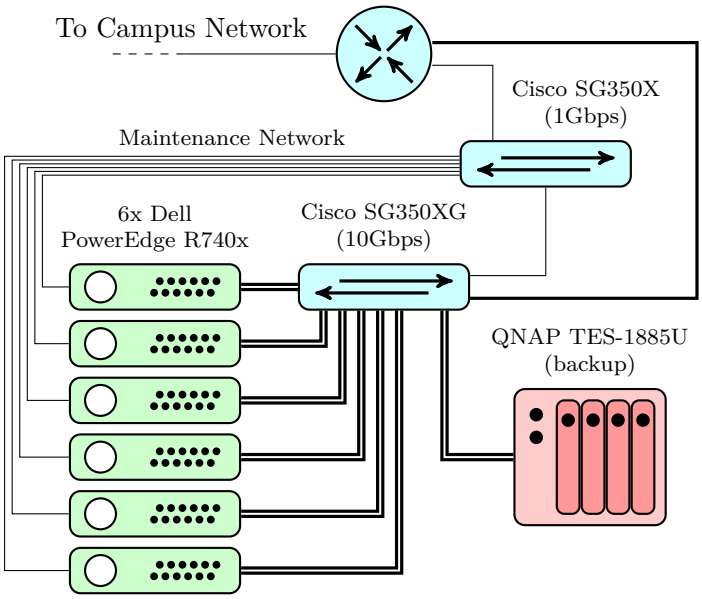

Figure 5.1: Physical infrastructure

All the tests and operations have been carried out on the cluster described above.

5.2 Measurements

The following section includes all the various tests, starting from the feasibility ones, to the production operation occurred in the September exam section.

5.2.1 Load tests

An initial test has been done by overloading one single machine (with 28 virtual CPUs) with 30 running PyCharm Instances which have been used by a group of volunteers.

The running Instances has been used in rather unusual ways, by running CPU intensive task or operations which caused noticeable video output operations. The following results emerged from this trial.

- The network load has proven not to be an issue: the most intensive video output operation never required more than 10 megabits per Instance (Fig. 5.3 shows the overall network usage during the exam, which never went beyond 200 megabits per second).
- The overall user experience was quite surprisingly not compromised, regardless of the high CPU load on the machine (due to the rather aggressive usage during the test).

5.2.2 Startup tests

To test startup times, 350 PyCharm Instances have been started, scattered across the whole cluster.

The Instance resources have been created from an automated script that has been run from outside the cluster. The Kubernetes API Server and the Instance operator processed all the requests (both coming from the computer that created the instances and from the Instance Operator itself) without accumulating delays or needing to queue requests.

The whole operation never exceeded 8 minutes: such time has been clocked from the creation of the Instances resources to the moment in which all the Instances became ready and accessible. Note that this phase does not include the PyCharm indexing phase (visible in the initial peak in Fig. 5.2) that

occurs right after the container startup; during the indexing phase PyCharm is partially unresponsive but still usable.

5.2.3 Delivery tests

Further tests have been done to better understand the behavior of the infrastructure during another critical phase, that is the final moment of the exam. During this time, the contents produced by the examinees are submitted and the load on the component that has to collect all the results can be high. Two solution have been tested.

NextCloud

NextCloud is the storage system which has been initially integrated in CrownLabs for letting students persistently store data inside Instances, allowing sharing them between other students and professors. It provides several libraries that enable programmatically working with its filesystem without too much programming effort.

This solution was used by setting the Submitter component so that it uploaded files in the Instance PVC onto a given NextCloud endpoint.

Such system however has proven to be too slow and unstable. Worst case tests with 350 instances have been run: the submission has been triggered concurrently on every instance, uploading 5 small text files (less than 2 Kilobytes in total) onto NextCloud. Said test completely paralyzed the NextCloud deployment inside CrownLabs cluster, that presents 3 replicas of the NextCloud server and 3 corresponding replicas of its PostgreSQL database. This mainly happened because of NextCloud needing to contact its database multiple times for each file operation.

Collector

The Collector component, initially developed for handling uploads for virtual-machine-based Instances, has been adapted to handle simple file uploads.

It worked flawlessly if compared to NextCloud: requests have been processed immediately as soon as the Submitter Jobs started. The same number of

Instances pushed the same files used to test the NextCloud based solution onto a single replica of the Collector without the need of queuing requests.

5.2.4 Production results

The final, overall number of students who signed in for such exam session was around 750.

To avoid overloading the system and managing possible overlappings, the exam has been divided into 3 rounds (i.e. 250 instances per round). Each round lasted 90 minutes, with 30 minutes between each of them. The cluster has been cleaned up from all active CrownLabs Instances after informing all the users subscribed to the service, then public access has been disabled.

For this operation, because of some limitations of the Moodle integration being incomplete at the time of the exam, all the Instances for the respective round have been started up upon each turn.

The resources limits have been set as follows:

- CPU has been limited to a whole virtual cores per each Instance, with 0.75 virtual cores reserved,
- Memory has been limited to 2 Gigabytes per each Instance.

The maximum load on the cluster has been detected only during the startup phase, given that the PyCharm application needs to perform intensive I/O operations during startup. The overall cluster usage in terms of CPU never exceeded 75 units out of 336 virtually available units.

Fig. 5.2 and Fig. 5.3 show respectively the number of CPU threads and bandwidth usage during the second round of the exam. Each line represents usage of each Instance.

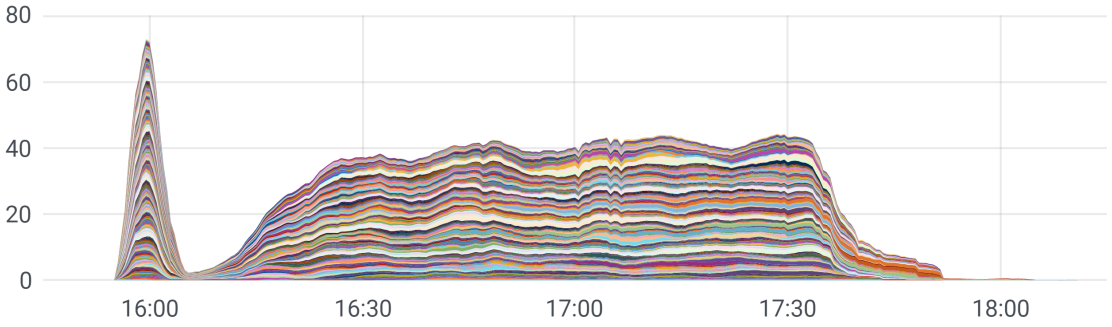

Figure 5.2: Overall CPU Usage on September exam session (turn 2)

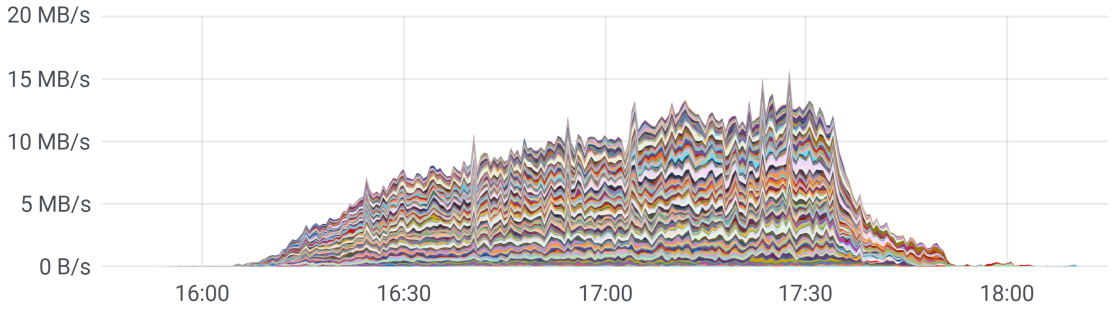

Figure 5.3: Overall bandwidth usage on September exam session (turn 2)

Chapter 6

Conclusions

The versatility of an infrastructure like the one behind CrownLabs has been crucial for making this work possible. The power of Kubernetes and its general purpose capabilities can be exploited to build complex applications without the need of writing critical procedures. Reliability and resilience are provided out of the box.

The tests have proven the effectiveness of the system, which provided a much more stable, usable and comfortable experience for the students, in comparison to CodeRunner. At the same time, it demonstrated the higher scalability of container-based Instances in contrast with virtual machines, especially in terms of startup time and resources usage.

Both professors and students from the Computer Sciences courses appreciated this solution: professor can build exercises with more freedom and without the concern of unstable environments; students, on the other hand, can work on the same environment on which they studied during laboratory experiences, without the limitation of custom solutions.

Further work is planned for better integrating the platform with the university exams infrastructure. Given the success of the September session, this solution has been chosen for the more critical winter exam session which will roughly expect ten times the number of students had on September.