POLITECNICO DI TORINO

Master's Degree in INGEGNERIA INFORMATICA (COMPUTER ENGINEERING)

Master's Degree Thesis

Delivering Remote Desktop Services to Remote Users: A QoE Perspective

Supervisors

Prof. Fulvio RISSO

PhD Marco IORIO

M.Eng. Federico CUCINELLA

Candidate

Guido RICIOPPO

October 2021

Summary

The 2020 pandemic accelerated some changes and introduced a lot of innovations destined to reshape long-lasting habits. As an example, while university exams and laboratory exercises evolved a lot in the method and contents, the laboratory and tools used to carry out them never changed in decades. Pandemic-related restrictions introduced the necessity to move services, tools, and resource-intensive programs from university laboratory workstations to students' personal devices.

CrownLabs was born to solve this intricate puzzle. The open-source project was started in late 2020 by a group of students at the Polytechnic University of Turing with the aim to provide a remote desktop environment to let students experience remote laboratories. It was extended in 2021 to make it suitable for a large number of users and to become a remote-exam platform.

This thesis work consists of an extension of the CrownLabs ecosystem with the intent to increase the overall remote-exams QoE. The developed observability extension enriches examiners' experience with extended metrics over the exam instances, while students are equipped with non-invasive tools to track at any time the connection towards the remote environment and its health status.

The work spans from the design of a monitoring infrastructure that perfectly fits the existing CrownLabs architecture, to the practical implementation and integration with multiple monitoring frontends. During the development phase, production-grade cloud technologies have been used, together with Git for open-source contribution processes.

After an initial test phase, the CrownLabs infrastructure, extended with monitoring features, has been used to carry out the Computer Science exam during the July and September 2022 PoliTO exam session.

Table of Contents

Chapter 1

Introduction

Organizing a university exam has become increasingly difficult in recent years: in addition to the usual logistical problems, modern university courses introduced new requirements to provide students with modern working environments.

It is often required for each student to access, during the exam, the same services, tools and programs used in practice workshops and home study, most of which are licensed and need powerful machines to run. Moreover, the lack of university laboratory workstations and COVID-19 pandemic restrictions introduced the necessity to move such exam tools to their same personal devices.

Most of the time, cloud technologies are the best solution to this intricate puzzle: they allow the delivery of remote desktop services to a heterogeneous plethora of devices by offloading the program execution, hence the computational power requirements, to remote servers. While lots of existing open-source and commercial Virtual Desktop Infrastructure (VDI) products provide a quick solution to the problem, they are far from being well-integrated with the wide exams ecosystem.

This thesis focuses on providing a QoE perspective on such remote desktop tools, increasing the observability of the remote exam environment for both users and maintainers through the implementation of ad-hoc monitoring tools.

1.1 Goal

The thesis has been developed at the Polytechnic University of Turin and consists on the extension of **CrownLabs**, a project started in 2020 during the *COVID-19* pandemic and developed by Ph.D. students and master's degree students in Computer Networks and Cloud Computing courses. The initial aim of the project was to enable students of the networking courses to continue on carrying out laboratory sessions in a remote manner but kept on growing and generalizing. In 2021 it was enriched with scalability capabilities, in order to support university exams with secure remote desktop environments.

CrownLabs exams QoE extension has been tested during the 2022 summer exam session in a "bring your own device" scenario, requiring students to connect to a remote desktop via a browser, either from home or from a classroom.

This thesis work aims on developing additional tools to support examiners and users with observability of the remote desktop environment status. By providing such feedback, examiners will enrich their experience with additional control over the exam instances, monitoring at different levels of detail the exam instances' health status. They will be enabled to bind each instance to an exam attendee and to track instances' network activities to avoid fraudulent behaviors. Students, on the other hand, will be equipped with non-invasive tools to understand at any time if their connection towards the remote system is healthy, and if they are making the best use of the available resources.

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 for this work.

The following design section summarizes how the general infrastructure has been set up and why.

The implementation section consists of 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 different metrics collected.

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

To provide CrownLabs monitoring capabilities to the different remote exam actors, a variety of tools and technologies have been used. Most of them concern cloud computing and distributed systems programming.

This chapter helps in understanding the outcome of the thesis discussed in Chapter 4, presenting some existing tools related to the problem statement and the different technologies and protocols on which the ad-hoc solution is based.

2.1 Related works

Providing monitoring capabilities on virtualized or containerized remote environments is not an uncharted problem. In one way or the other, every container or virtual machine orchestration technology provides tools to observe the overall system status in varying degrees of detail: from the overall server hosting the workload to the container or VM instance executing a job. CrownLabs is not an exception: the cluster maintainers have access to tools to monitor a variety of resources on different details.

However, such tools are often unavailable to the service users, and in case of an exam, to the examiner. Even if they were, moreover, they would most of the time be incomprehensible, being designed for cloud and system administration experts.

The following sections 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 form of remote experience.

2.1.1 VLAIB

Polytechnic of Turin technical department developed Virtual LAIBs (VLAIBs) on top of an on-premise cluster equipped with VMware Horizon [1]. Its purpose is to provide the same environment present in the terminals offered by 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 product license, compatibility, or performance issues that students might encounter.

VMware Horizon administrators can monitor their end-user computing environments using the **ControlUp** add-on. The framework provides a comprehensive dashboard for real-time monitoring, and scripts can be instantiated to proactively avoid or reactively remediate issues with remote desktops and hosting servers.

Such a complete tool, however, is available only for university system administrators and it is not extensible to provide the same metrics to the end-users.

2.1.2 Prometheus and cAdvisor

The remote-desktop environments provided by CrownLabs are based on Kubernetes: an open-source orchestration platform working with containers (see Section 2.6. Kubernetes offers some built-in monitoring capabilities, including *cAdvisor*: an open-source agent that monitors resource usage and analyzes the performance of containers. Kubernetes requires such metrics to actively monitor the health of containers (Liveness and Readiness Probes) and to automate Pod scalability (Horizontal Pod Autoscaler).

Cadvsor provides metrics with variable frequency and with a customizable

[1] https://www.vmware.com/products/horizon.html

level of granularity: CPU, memory, and network metrics can be obtained down to the level of a single container.

Metrics providers like cAdvisor can export data as *Prometheus metrics*[2] Database entries, messages, or via a REST API. In that case, a Prometheus metrics server will scrape and store metrics on a stateful Time Series DB. The collected metrics can be consumed by a variety of clients, the most famous being Grafana.

The union of cAdvisor rich metrics and Prometheus may seem the perfect solution to the problem stated in Section 1.1: a student-oriented client ad an examiner dashboard may consume the stored metrics offering an ad-hoc solution using mainly Kubernetes built-in tools. Unfortunately, cAdvisor does not fit the real-time feedback requirement. It periodically gathers container stats, but the collection interval can not be personalized. Moreover, the same interval is dynamically adapted depending on how active the watched container is, in order to reduce resource usage. Intervals may vary from 4s up to 20s[3].

2.1.3 Commercial products

Several commercial products can be deployed on container orchestration platforms in order to discover, map, and monitor applications and microservices. Most of them can be integrated directly on public cloud provider platforms, such as Amazon Web Services(AWS) or Microsoft Azure, or can improve consolidated technologies in the same field as Prometheus.

It is likely but not certain that mixing the capabilities of public cloud providers with commercial monitoring tools would have allowed achieving the same outcome of this thesis. Most public cloud providers offer personal environments remotely accessible and most monitoring frameworks can be accessed by rich REST API, enabling infinite client solutions.

Such an approach, however, would have had a low impact on the open source community and would have required the purchase of several paid licenses.

[2]https://prometheus.io

[3]https://github.com/google/cadvisor/issues/2660

Dynatrace

Dynatrace uses OneAgent, a unique agent to collect and unify performance metrics for servers, containers, applications, services, databases, and more. It is integrated with most of the leading enterprise cloud ecosystems that support dynamic container orchestration.

Datadog

Datadog is a wide framework that offers infrastructure, network, and container monitoring. It automatically discovers and provides real-time visibility of containerized environments.

2.2 Virtual Machines VS Containers

The typical approach to provide remote users with software or platform services is to instantiate a Virtual Machine (VM) on a server. A VM is a software compute resource running a full-fledged "guest" Operating System (OS) on top of a physical "host" machine. VMs have historically been used for server virtualization, making the most of their computing resources by their consolidation. From the user perspective, QoE is guaranteed by the transparency of virtualization: the experience is not different from using a regular Operating System on a physical machine.

When it is the case to access the guest OS remotely, virtualization is performed by a Virtual desktop Infrastructure (VDI) that takes care to implement desktop virtualization with Remote Desktop connection Protocols discussed in Section 2.3. Usually, the guest OS is initialized with uncontrolled root access, so additional software is required to handle security, alongside the main software the VM was intended to be shipped with.

VDIs fit the requirements needed for remote exams and laboratories, either from home or with a bring your own device (BYOD) approach. In fact, the same CrownLabs was initially born to run remote environments based on VMs.

Unfortunately, despite of being easily usable and flexible, VMs present several drawbacks, especially related to performance and scalability. One of the major issues is given by the overhead that virtual machines need in order to

be functional. Allocated CPU and memory resources can become conspicuous when the infrastructure needs to scale 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 consists of process isolation 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 isolated process or external ones.

By eliminating the entire guest OS and reusing the existing host kernel, only needed system processes are left alive (see Fig. 2.1), significantly reducing the resource allocation overhead.

With scalability in mind, CrownLabs support for remote exams delivers container-based instances, each running the main application only.

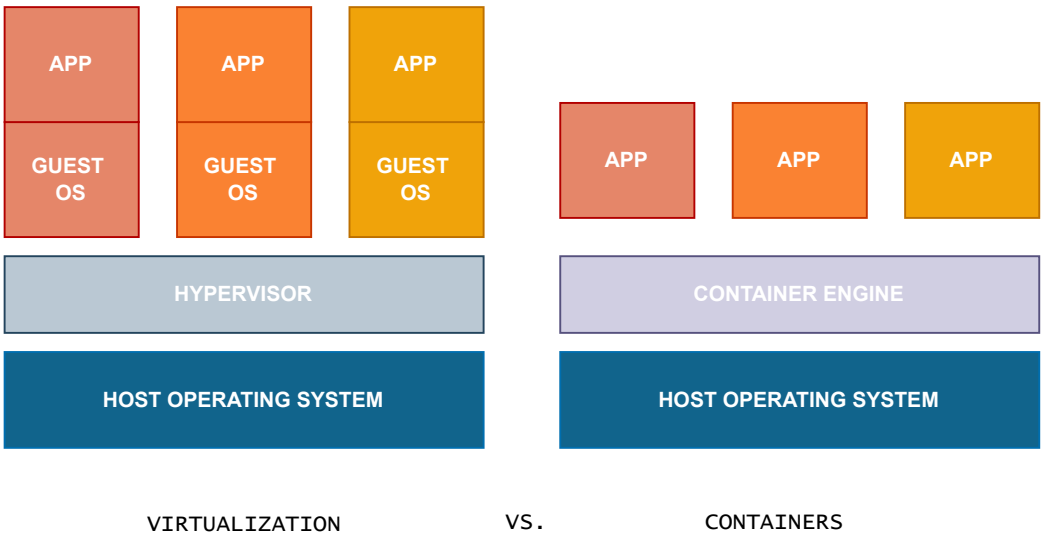

Figure 2.1: VM vs Container architecture

2.3 Remote Desktop Protocols

Regardless of the technology used to build the virtualized desktop environment, the protocol which enables the actual remote control is a common point. 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 resource costs.

2.4 Container isolation by means of Cgroups

The two Linux technologies that make up the foundation of building and running containers are **cgroups** and **namespaces**. This section focuses on cgroups (short for control groups), being the technology handling accounting and the distribution of processor time and memory between tasks.

When a container is created, it has potential access to all physical resources of the server (i.e. CPU, memory). Without some form of limitation, a single containerized process could tear down an entire server simply by allocating a big chunk of memory or snatching CPU time from other processes. Cgroups allow isolating containers placing the specific processes in control groups embedded in the system. This is done by defining *resource limits* that regulate the access to the overall system resources.

Cgroups also take care of accounting: every time a process confined in a group allocates some memory or uses CPU time, an accounting subsystem registers the usage statistics on the related control group files. This is why every

monitoring tool, like the one discussed in Section 2.1.2, directly or indirectly uses cgroup subsystem information to retrieve usage statistics to provide to clients as metrics.

2.5 Docker engine and Dockershim

Docker is one of the most popular tools for application containerization. It is based on a client-server architecture: a Docker *client* talks to the Docker *daemon* called *dockerd*, which interacts with the Linux host to build, run and manage containers.

The docker daemon itself is in charge of collecting the metrics regarding the managed containers. It supports CPU, memory usage, memory limit and network IO metrics. Those metrics can be accessed either from a CLI or from the provided SDKs for different programming languages, like Go and Python. The SDK allows developers to create custom monitoring systems that can be integrated natively with other services.

It is worth noticing that despite being Kubernetes born initially upon the Docker container runtime (see Section 2.6), the two tools are not compatible anymore. To use both, Kubernetes was integrated with a piece of software shim called *Dockershim*, filling the gaps of incompatibility.

Referring to the date of this writing (2022), PoliTO CrownLabs' deployment is based on a Kubernetes cluster using Dockershim.

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[4] and enable it to host cloud native software.

Kubernetes practically consists of an orchestrator for containers[5]: through

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

[5] Thanks to additional software like KubeVirt, which is used also in CrownLabs from its

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) part of 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 represents a 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 with no issues, while namespaced resources need to be part of the same referring namespace, 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.

---

beginnings, it is possible to schedule and run also virtual machines instead of just containers.

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 the 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 with 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 assigned 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) that 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 with 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), and 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 provides 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*[6]: they provide 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 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 up 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 stores for Pods.

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

- *Persistent Volumes Claims* (PVCs) instead are native Kubernetes (namespaced) resources that 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 rebuilding 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

embedded into the Kubernetes engine itself, it is possible to 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 must be compliant 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

etcd is a consistent and highly-available key value store used as Kubernetes' backing store for all cluster data

When deployed with a distributed 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 the 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 exploits 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: the 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 with 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 resource 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's main goal is to run the *reconciliation loop* for the default Kubernetes resources. Such a process is the procedure that drives the 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 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. It operates by requesting to the underlying Container Runtime API the creation, modification or creation of a container.

This component is also a node agent for managing container resouces, exposed by the kubelet API endpoints.

`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 of 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.

2.6.3 Kubernetes resource metrics pipeline

Kubernetes embeds a system able to offer a basic set of metrics, used to support automatic scaling and similar use cases. Indeed the *Metrics API* and the metrics pipeline that it enables, only offers the minimum CPU and memory metrics to enable automatic scaling up.

Basically, the metrics pipeline is composed of a *metrics-server* deployed as a Cluster add-on component, that collects and aggregates resource metrics pulled from each kubelet. The aggregated metrics are finally exposed by the Metrics API.

As mentioned in Section 2.1.2, each kubelet refers to the cAdvisor daemon to collect container metrics. This makes it difficult to use the metrics-server to implement a personalized monitoring tool, since the scraping interval and variety of collected metrics are not flexible as needed.

2.7 CRI-API

At the lowest layers of a Kubernetes node is the Container Runtime, a piece of software that, among other things, starts and stops containers. In the early days of Kubernetes, the only supported container runtime was Docker Engine. Not much later, new container runtimes began to emerge and gain notoriety.

To allow flexibility, the CRI (short for Container Runtime Interface) was released. It is a plugin interface that enables the kubelet to use a wide variety of container runtimes, without having a need to recompile the cluster components. Each node of a cluster needs a working container runtime so that the kubelet can launch Pods and their containers.

The CRI defines a standard API that allows Kubernetes internal or external clients to connect the container runtime via gRPC.

2.7.1 CRI Stats

In Kubernetes 1.8, the CRI-API was enriched with new functions to let the kubelet get container stats from the CRI. This move is intended to reduce the dependency of the kubelet from cAdvisor, since it is a separate open-source project and requires each container runtime to add the additional corresponding package to support tracking metrics.

With CRI being the new standard, it was a natural progression to augment the API to serve container metrics to eliminate a separate integration point.

However, up until Kubernetes 1.24, the Kubelet still relies on cAdvisor to track resource usage at the pod level. It only uses the CRI-API to obtain container-level metrics. The API supported today includes the two gRPC endpoints `ContainerStats` and `ListContainerStats` to gather cumulative memory and CPU usage of a set of containers.

This particular enrichment of the API opened a wide range of possibilities for cloud-native developers, since it is an effective and well-designed way to obtain rich container metrics at the desired frequency.

2.8 PoliTO Exam platform

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

2.8.1 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,s and other e-learning projects in schools, universities, workplaces and other sectors.

The learning system was integrated into the Polytechnic University of Turin infrastructure to enable computerized exam management. Moodle, in fact, 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.

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

The implementation of Moodle with CrownLabs allowed the evolution of 2022 post-pandemic physical exams to a Bring Your Own Device modality,

delivering the same LAIBs services through containerized instances. This upgrade made it possible to remove the limit of simultaneous exam participants due to the limit of LAIB workstations.

2.8.3 Remote exams

The lockdown which began with the 2020 pandemic and the subsequent prevention measures taken by the Italian 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 on students' computers in order to avoid cheating and distraction. Subsequently, the Moodle infrastructure needed to be improved to support a larger number of sessions.

Subsequently, from 2021, Moodle was enriched with a plugin that enables the instantiation of CrownLabs containerized instances in support of students. The implementation allowed exam participants to come back to the experience delivered through LAIBs computers, but bringing the services to remote-desktop environments instead of physical ones.

2.8.4 Existing monitoring tools

The implementation of the PoliTO exam platform with CrownLabs capabilities also comes with a basic form of cluster monitoring tools. These tools are not intended to be used by examiners or students, but are an effective way to provide system administrators with the information to observe communications and interoperability between various nodes of the cluster.

CrownLabs monitoring is achieved with the following set of instruments.

Prometheus already discussed in section Section 2.1.2.

Thanos is an open-source extension of Prometheus storage format that stores cost-efficiently historical metric data in any object storage while retaining fast query latencies.

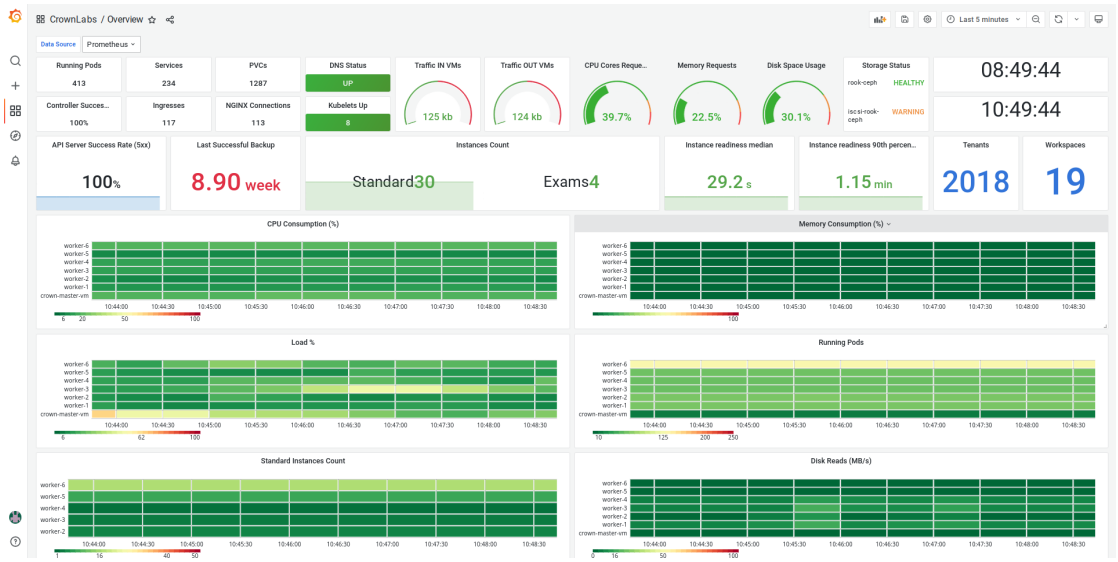

Figure 2.2: Example of Grafana Dashboard in CrownLabs

Grafana is an open-source platform for monitoring and observability. It glues the information scraped by Prometheus and stored by Thanos allowing the user to query, visualize, and explore metrics. The visualization is provided by creating dynamic and reusable dashboards (see Fig. 2.2).

While being effective for the cluster developers and administrators, those metrics tools are hardly accessible and understandable for sporadical users.

Chapter 3

Design

Increasing QoE when dealing with remote-desktop environments used to deliver online exams is a quite general problem. Consequently, a solution may be based on a general component, despite the different implementations.

Yet, most of this thesis work focused on the design of an architecture that could fit inside a bigger existing project called **CrownLabs**. In order to understand some design choices, it may be useful becoming familiar with the general architecture that enables “cloud grade“ remote exams.

This chapter starts with an overview of the Kubernetes backend used by CrownLabs infrastructure, focusing thereafter on an architectural extension that aims to improve the monitoring system in the context of mass-scale remote exams.

3.1 CrownLabs operators-based infrastructure

From the user point of view, CrownLabs is not different from other services accessible from a web browser: a dynamic web page uses a specific API to control the backend implementing the business logic. However, CrownLabs infrastructure is by no means “traditional“: the entire system is built up using components from the Kubernetes ecosystem, extended with custom software implementing the core business logic.

The frontend page provides access to Custom Resources interacting directly with the Kubernetes API server: it allows final users to spawn new environments, and connect to their instances, similarly as `kubectl` does from CLI (see Section 2.6.2).

Following a design aimed at automation, the application components' integration directly into the Kubernetes infrastructure is done by means of CRDs (Section 2.6.1). The business logic, instead, is implemented according to the *operator pattern*.

Fig. 3.1 shows an overall view of CrownLabs' frontend and backend design. It is possible to notice how the architecture is composed both of standard Kubernetes components and extensions managed by operators.

Operator pattern The Operator pattern aims to capture the key aim of a human operator who is managing a service or set of services. The operator software implements a custom Kubernetes controller to control the life-cycle of custom resources on behalf of Kubernetes users.

3.2 Kubernetes-powered back-end

The management and development complexity of CrownLabs backend is reduced delegating as many tasks as possible to the Kubernetes infrastructure. The main *entities* involved in the CrownLabs infrastructure are mapped to Custom Resources, so that the user can interact with them by sending requests directly to a Kubernetes API server, either via `kubectl` command or using the graphical interface provided on the website.

Regardless of the used client, users can experience remote computing performing the following operations:

- 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.

In addition to the latter general-usage operations, a new exam-specific workflow was recently added. Using the moodle plug-in, a student can:

- create an exam-binded Instance with a persistent working directory;
- access the Instance from the Moodle interface via a special type of question within the quiz;
- stop Instances by terminating the quiz. This same action triggers a submitter Job that collects the content elaborated by the student and uploads it to a specific endpoint;

3.2.1 CrownLabs Resources

The following section illustrates the main custom resource 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 with it. Each environment can be either graphical or text-based. In the second case, it is generally reachable through ssh.

Templates A Template can be seen as a "model" that describes what an instance should consist of. Its specification holds information relative to the image to be used in the instances created from that template, if 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

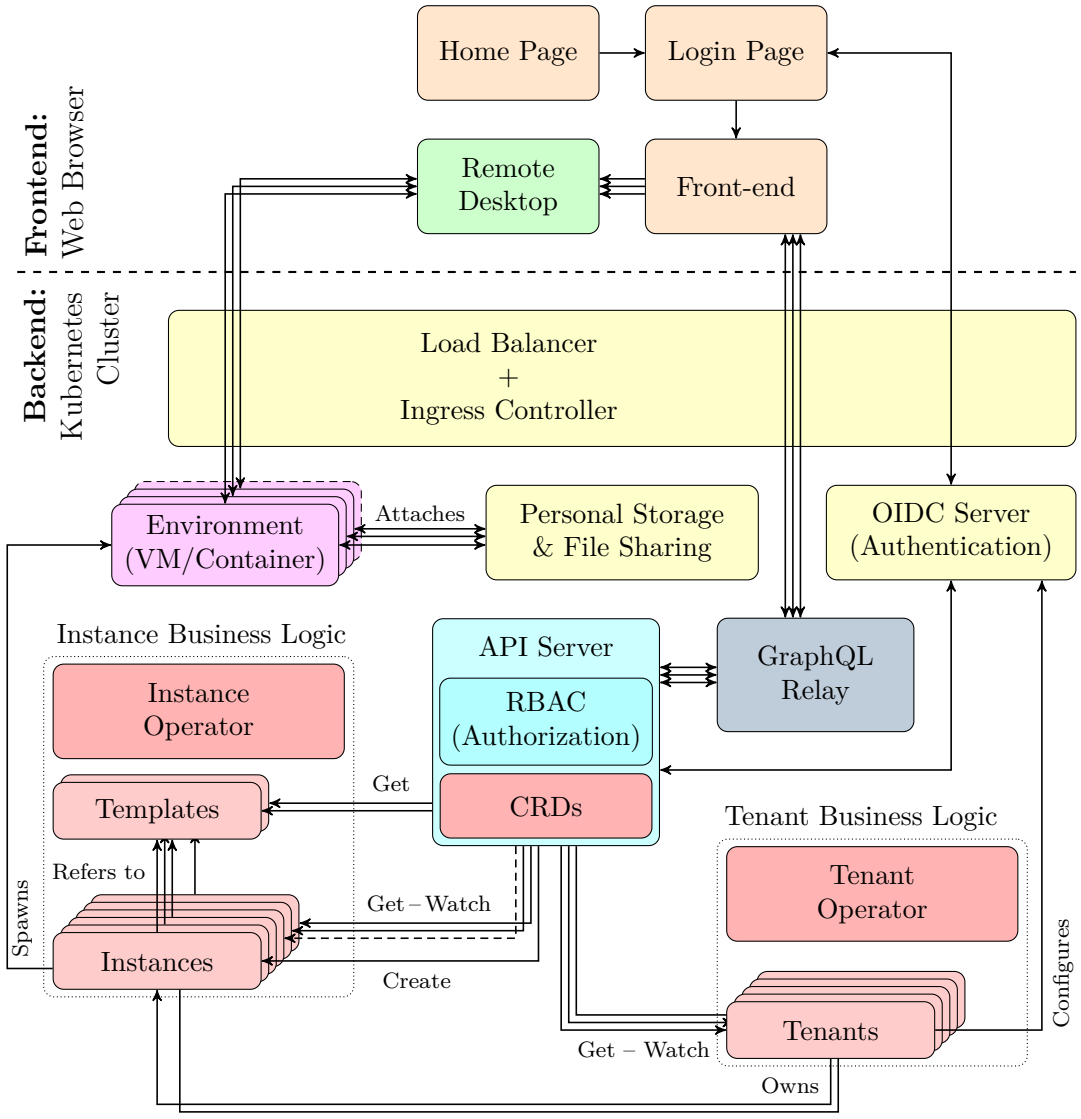

Figure 3.1: Overall schema

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 with them, so 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.

Containerized graphical instances

An Instance component architecture is composed of multiple elements. In order for the environment to be served on a remote-desktop page, the single *Application container* is not enough. Two other support processes are needed: a *Display Server* that hosts the desktop environment on which the application window is hosted, and an intermediate component (proxy) between the browser and the display server arbitrating the bi-directional communication. The latter is needed because of the sandbox nature of web browsers.

Fig. 3.2 shows the described architecture. Further implementation details can be found in Chapter 4.

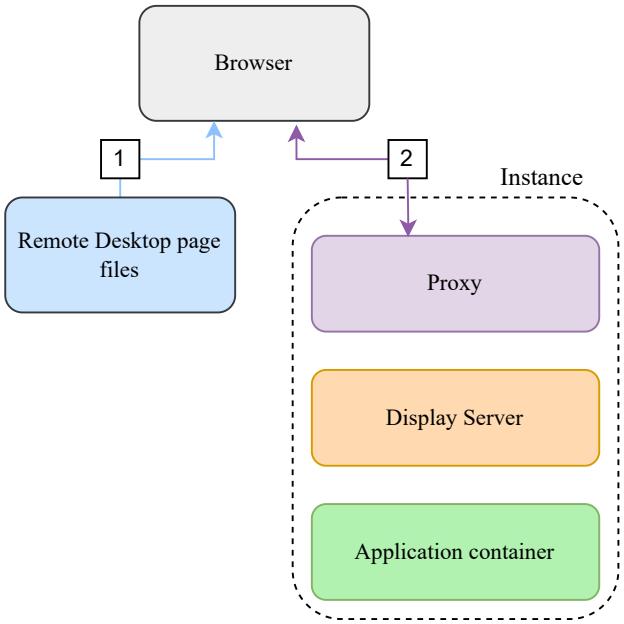

Figure 3.2: Containerized instance architecture

3.3 Remote desktop management

In the 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 almost complete integration with the keyboard (some combinations of keys would interfere with both the client operating system and browser and cannot be directly forwarded to the remote environment).

3.4 Exam Agent

As already mentioned, CrownLabs capabilities have been integrated with Moodle throughout a plug-in in order to deliver remote exams to Polytechnic of Turin students. This section illustrates the exam infrastructure that is put beside an existing CrownLabs deployment.

Landing

The Landing component is an interface between the Kubernetes cluster and Moodle. It is also called *examAPI*. Students are not allowed to access directly the Instance from the quiz page. Instead, when they try to access the Instance, they contact the Landing component:

- if the Instance related to the student is ready, the user is redirected to the remote desktop environment;
- if the Instance is starting, a loading spinner is displayed to the student until the instance becomes ready;
- if the Instance does not exist, the instance is created on demand;

The *examAPI* is an interface that opens infinite gates to future exam-related tool implementation, since it can be used to obtain a list of running instances related to the exam, create new ones or delete existing ones. It also provides information about the student bound to each instance, the exam termination date, and if the persistent folder has been already submitted.

Content downloader

When an Instance is created from a Template, this component is in charge to populate the persistent folder on which the student is supposed to work on. Examples of downloaded content are a PyCharm project folder or a Blender library.

Terminator

The Terminator is a daemon running a control loop during the entire duration of the exam session. Once the Moodle attempt is closed, the Terminator is in charge to stop the associated instance.

Submitter

This component includes a Job dedicated to the collection of the content of each persistent working folder and then upload to a specific endpoint defined in the Template of the Instance.

3.5 Exams monitoring infrastructure

Despite the integration between an exam management system and CrownLabs remote computing capabilities, the whole architecture lacks a monitoring component that could help students and examiners to obtain an overall view of the system.

This thesis work focused on the design and implementation of such components, in order to improve the QoE of the different exam actors. The following sections illustrate some of the design choices made as a result of the discussed requirements.

Fig. 3.3 shows an architectural overview of the monitoring infrastructure.

3.5.1 Metrics collection

The first step to provide observability to users is the retrieval of information regarding the environment they are working on. In the context of CrownLabs

exams, the working environment is an Instance for the student and the list of instances with their exam-related status for the examiners.

The monitoring system was designed to use the information already present in the infrastructure as much as possible. For this reason, there was no need to reinvent a component binding exam-related information to the right Instance, since the *Landing* component already provides a complete overview of the state of exam Instances.

Taking into consideration the state of an Instance, instead, a lack of information emerged: there is no high-level component already existing in Kubernetes or implemented in CrowLabs able to provide real-time statistics about an Instance. Moreover, existing services doing a similar job do not fit inside the existing CrownLabs exam architecture, so a set of new components has been designed and developed.

Instmetrics

An *Instmetrics* server (short for Instance Metrics Collector) has been added to the existing architecture. This component is a daemon watching for existing containerized Instances and in charge of scraping the stats for each Application container belonging to a containerized Instance (see Fig. 3.2).

Being a daemon, at least one Instmetrics server is required for each node of the Kubernetes cluster. Which metrics are collected for each Application container depends on the information available on the Kubernetes backend; later in the implementation chapter a possible CrownLabs-compatible approach will be discussed.

The Instmetrics server will serve metrics to the proxy component of each containerized Instance: this will form a **service chain**.

3.5.2 Metrics aggregation

It is useful for metrics clients to rely on a single source of truth, instead of collecting information from different sources on their own. This also helps with accounting and regulating the access to monitoring features, which often contain sensitive information like a connections register.

Metrics aggregator

A metrics aggregator is located inside the *Proxy* component of each Instance: it connects to the Instmetrics server to enable monitoring on the specific instance. The component increases the transparency of the CrownLabs backend architecture: the client performing monitoring can retrieve metrics from the same endpoint used to connect to the remote desktop environment of the Instance. The retrieval of metrics can be activated or disabled by an option after the connection to the Instance.

An important aspect intended to increase Instance observability is the monitoring of network performance and connections. In this regard, the metrics aggregator itself is in charge to track connections to the remote environment and analyze network parameters for each active connection.

The overall Instance metrics will be provided both to the remote-desktop users and to a summary dashboard used by the examiners. Regarding the type of client, separation of concerns must be performed: the student is interested in his connection stats only, while the examiner needs wider visibility. It could be useful to track the source of each connection and details about concurrent connections to the remote environment.

3.5.3 Frontend monitoring tools

The applications of monitoring tools are potentially endless: once the metrics of interest are obtained, they can be represented on different frontends in a way that fits the user.

For the sake of this thesis work, the actors interested in enriching visibility over the Instance or the wider system are:

- **students** actively making use of the remote desktop environment. They want to be aware of the real-time usage of the Instance resources and their connection quality;
- **examiners**, often professors, monitoring the exam course. They want to know how every Instance is behaving and who is connected from where;

Remote desktop add-on

A view can be provided to students in several ways: a new browser page, a metrics strip on the same remote desktop page, or an extensible add-on that can be closed when not used.

The design choice, in this case, fell on the add-on tool. This allows students to be always focused on the same page, avoiding distractions. It is important, still, to avoid the additional tool being invasive on the exam page; this is why it was decided to design a draggable element, extensible when required.

Also, a discussion about the content of the add-on was held. Usage metrics are subject to high variance; care must be taken not to attract student attention when not necessary. This is why the implementation following this design choice will require a study of typical users' behaviors, making a distinction between real critical situations and false alarms.

Examiner dashboard

The examiner dashboard has been designed as a single-page application accessible from the exam management system. It is wanted to guarantee a seamless experience to the dashboard users as well: the monitoring page must be linkable from Moodle, giving the perception of being an additional Moodle plugin provided by CrownLabs.

The dashboard is composed mainly of a tabular layout in order to strictly every piece of information in a defined space. This decision was made in order to avoid users drowning in information. At PoliTO, exams with more than 200 booked students are not uncommon; in such cases, relevant information shouldn't be overshadowed by the number of rows.

The table must show brief stats about each instance: punctual resource usages, for example, are not relevant. Examiner attention must be triggered only when trend values of resources go wild or unexpected connections are taking place. Having the possibility to sort columns is also useful.

Basically, information of interest for examiners during the exam can be summarized in the following columns:

- student name and its university ID;

- Instance overall status. This field must trigger the examiner's attention if the CPU, memory, connections, or network are having a long-standing warning state.
- CPU usage trend. Providing instantaneous values when required may be also useful;
- memory usage trend. Same as CPU about the real-time usage;
- network connections quality. This field must fall in a warning state when the connection's Round Trip Time does not guarantee a smooth experience for Instance users;
- active connections number and their classification. Since the user may be not familiar with IP addresses, each address must be labeled and associated with a source location. Providing the connection time may be also useful;
- connections history. It must be a register of active or past connections;

The dashboard should provide also the possibility to jump to each specific remote desktop environment as a spectator.

3.5.4 Microservices approach

The components described above enable observability over Instances implementing a **service chain**. This approach follows the fundamentals of the Microservice architecture pattern:

- being CrownLabs an open-source project, multiple developers worked on the application;
- new developers quickly become productive, since the introduction of a new feature requires understanding only a small subset of existing components;
- services in the chain are loosely coupled: they communicate via the network and each component has very little knowledge about others' inner logic;
- services are highly coherent: each component relies on others in order to deliver a feature. For example, the Proxy element inside an Instance is not able to provide utilization metrics to the remote desktop or the Dashboard by itself: it requires a running Instmetrics server;

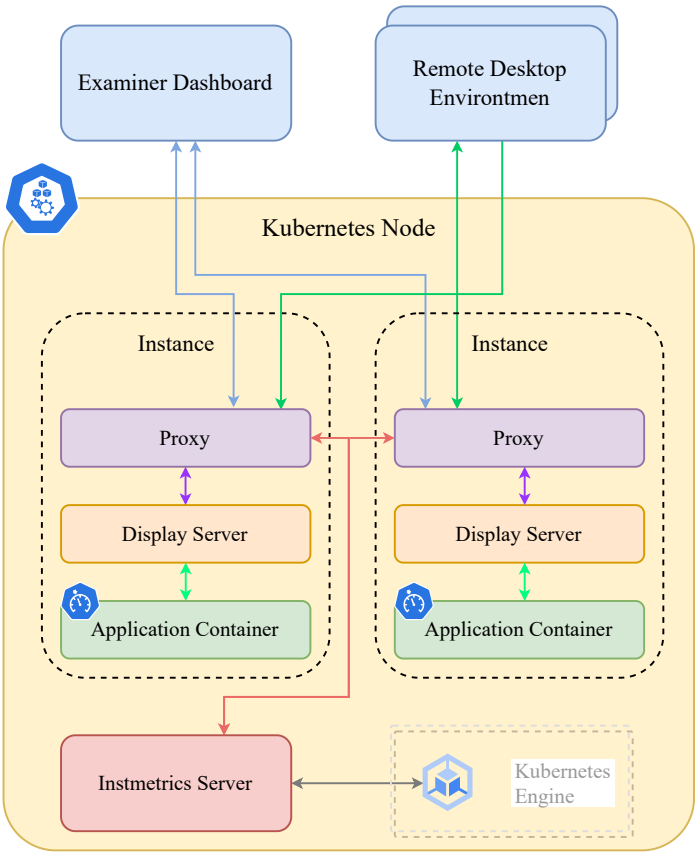

Figure 3.3: Monitoring infrastructure overview

Chapter 4

Implementation

Defined an architecture for the **exams monitoring infrastructure** in CrownLabs, the thesis work focused on implementing the needed components.

Despite the requisites being well defined during the design phase, the following implementation was definitely not linear: in Kubernetes, the same feature is often provided on different levels of abstraction. The more abstract the development interface, the easier to understand but the harder to personalize.

4.1 Instmetrics server: metrics scraper

The first step in developing a monitoring service is the retrieval of metrics. The implementation depends on which information needs to be collected and how frequently.

In order to provide a rich QoE to different actors of CrownLabs exam infrastructure, the following requisites have been defined:

- application container stats:
  - CPU usage;
  - Memory usage;
- connections tracking:

  - connections source;
  - active connections counter;
  - total connections counter;
- customizable metrics update granularity up to 1s;

The *Instance Metrics Collector server* (abbreviated as *Instmetrics*), is responsible for the collection of containers' **CPU** and **memory** stats, with customizable frequency.

This section deep dives into implementation details and describes the tools used to implement the *Instmetrics* service.

4.1.1 Scraping metrics from CRI-API

What is CRI

The *Container Runtime Interface* (presented in Section 2.7) enables the Kubernetes *kubelet* to use a wide variety of container runtimes [1].

Currently, CrownLabs is deployed in the Polytechnic University of Turin upon two Kubernetes clusters paired to form a multi-cluster using Liqo [2]. The first cluster nodes use *Docker* as container runtime with *the Dockershim* component for CRI compatibility, while the other one relies upon *containerd* as the container runtime, eliminating the middle man.

Through CRI introduction, this kind of flexibility is made possible, where the same Kubernetes-based software (like CrownLabs) can use a wide variety of container runtimes without the need to recompile.

However, knowing the infrastructure under the abstraction was recommended during development, since the plugin API of CRI does not provide always a seamless experience.

Fig. 4.1 shows how Docker actually uses Containerd as the underlying

[1] A container runtime is the process that does the actual work of creating, running, and destroying containers.

[2] https://liqo.io

container runtime. Switching to Containerd as a container runtime eliminates two layers of intermediation.

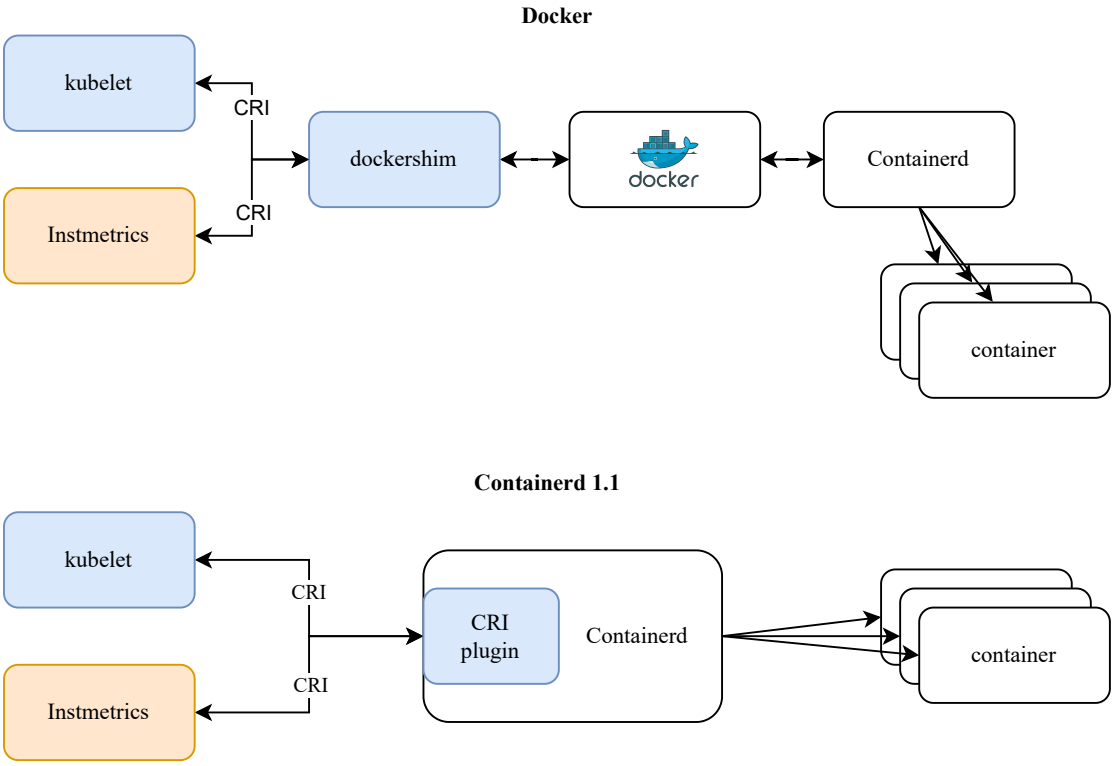

Figure 4.1: Dockershim vs. Containerd CRI Architecture

Services provided by the CRI-API

The CRI-API includes two *gRPC* services (more on gRPC on Section 4.1.2): `ImageService` and `RuntimeService`. The ImageService provides *Remote Procedure Calls* (RPC) to pull an image from a repository, inspect, and remove images. The *Instmetrics* uses the second service, called `RuntimeService`, which contains RPCs to manage the lifecycle of the pods and containers, as well as calls to interact with containers (exec/port-forward/**retrieve stats**).

Below are some exposed RPCs in `protobuf` format.

Listing 4.1: Runtime Services exposed by CRI-API.

```
service RuntimeService {
    // Sandbox operations.
    rpc RunPodSandbox(RunPodSandboxRequest) returns (RunPodSandboxResponse) {}
    rpc StopPodSandbox(StopPodSandboxRequest) returns (StopPodSandboxResponse) {}
    rpc ListPodSandbox(ListPodSandboxRequest) returns (ListPodSandboxResponse) {}

    // Container operations.
    rpc CreateContainer(CreateContainerRequest) returns (CreateContainerResponse) {}
    rpc StartContainer(StartContainerRequest) returns (StartContainerResponse) {}
    rpc StopContainer(StopContainerRequest) returns (StopContainerResponse) {}
    rpc ListContainers(ListContainersRequest) returns (ListContainersResponse) {}
    rpc ContainerStatus(ContainerStatusRequest) returns (ContainerStatusResponse) {}

    // Container metrics operations.
    rpc ContainerStats(ContainerStatsRequest) returns (ContainerStatsResponse) {}
    rpc ListContainerStats(ListContainerStatsRequest) returns (ListContainerStatsResponse) {}

    // Sandbox metrics operations.
    rpc PodSandboxStats(PodSandboxStatsRequest) returns (PodSandboxStatsResponse) {}
    rpc ListPodSandboxStats(ListPodSandboxStatsRequest) returns (ListPodSandboxStatsResponse) {}
}
```

In the case of Instmetrics, the most important information is provided in `ContainerStatsResponse`, which contains:

- cumulative CPU usage gathered from the container. The value is measured in cores usage nanoseconds;

- memory usage gathered from the container. This metric is of gauge type;
- usage of the filesystem writable layer;
- a list of container attributes;

4.1.2 Implementing a CRI-API client

CRI consists of a *Protocol Buffer* and *gRPC API*. To exploit the exposed services, the *Instmetrics* (as the Kubelet) communicates with the container runtime over Unix sockets implementing a gRPC client (Fig. 4.2).

Remote Procedure Call

Remote Procedure Call (RPC) is an inter-process communication technique that allows nodes in a distributed environment to execute a subroutine in a different address space (usually another computer). The *client* accesses the services or resources as they are local, easily calling a function.

In reality, however, the *server* providing the services is the only one implementing the logic of the procedures. After receiving a request whit the related parameters, it computes and sends a response back to the client over the network.

In order to simulate the implementation of actual procedures, the client contains a *stub* of code that represents the remote procedure code. When the procedure call is issued, the stub receives the request, and the related middleware forwards it to the remote server.

The specification of provided services and the structure of the resources are described by an Interface Definition Language (IDL). Client and server may be running on different operating systems and using different programming languages. That is why a common language to describe shared structures is required.

gRPC framework

gRPC is a robust open-source RPC framework used to build scalable and fast APIs. Its development was started by Google in 2015, which needed an RPC framework to link many microservices created with different technologies.

Protobuf or Protocol buffer is the Interface Definition Language used by the framework. It includes a serialization/deserialization protocol and enables the easy definition of services and auto-generation of client libraries. gRPC services and messages (see the example in Section 4.1.1) are defined in proto files.

Protoc is a Protobuf compiler that generates client and server stub code.

gRPC relies on HTTP/2, which supports binary framing, bidirectional full-duplex streaming, and flow control mechanisms.

While HTTP natively supports mediators for edge caching, gRPC calls use the POST method, hence the responses can't be cached through intermediaries. However, to overcome this problem in the CrownLabs exams monitoring infrastructure, both the Instmetrics (gRPC server) and Websockify (gRPC client) implement custom caching mechanisms.

4.1.3 Docker Engine API

As mentioned in Section 4.1.1, the current CrownLabs deployment uses both Dockershim and Containerd as container runtime.

The first development iteration of the *metrics scraper* component inside the Instmetrics foresaw the use of the CRI-API only for metrics collection. Unfortunately, despite being compatible with CRI, Dockershim does not fulfill all the requirements as expected: not all the fields of `ContainerStatsResponse` are correctly populated.

This inconvenience required an ad-hoc implementation in the case Docker is used as the underlying container engine. In such cases, container stats are extracted using the Docker Engine API to interact with the Docker daemon.

Being the Instmetrics service written in Go, the Docker Go SDK has been used. Depending on whether Dockershim runtime endpoint is provided or not, the `DockerMetricsScraper` or the `CRIMetricsScraper` will be used by the service.

Yet, a connection to the CRI-API is still required, since the service needs to identify an Application Container ID from the Pod ID provided by the Instmetrics client.

Fig. 4.2 shows the different elements and communication protocols composing the Instance Metrics Collector infrastructure.

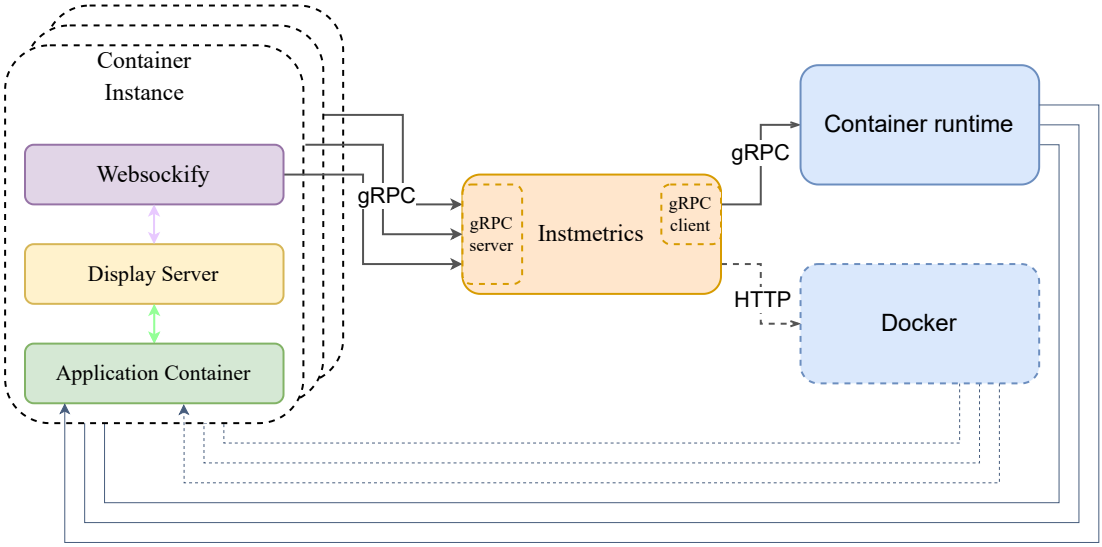

Figure 4.2: Instmetrics infrastructure

4.1.4 Caching mechanism

Since gRPC does not natively include caching mechanisms, each request for `ContainerMetrics` from the client results in the execution of a procedure server-side. One single Instmetrics server is deployed in each Kubernetes Node, so the server is potentially subject to a high number of requests in short timeframes.

For this reason, a caching mechanism has been implemented inside the `Metrics Scraper`: metrics are collected from CRI or Docker always with the same frequency, ideally in the order of `1s`, and results are saved on a state `Map<PodName, Metrics>`. When a request arrives, a response is assembled from the cached information.

4.1.5 Instmetrics exposed API

An Instmetrics server exposes collected metrics through a gRPC API. The service listens for gRPC connection on the (personalizable) port `9090` and

responds to metrics requests given a specific Pod name.

Listing 4.2: Services and Resources defined by the Instmetrics component.

```
service InstanceMetrics {
    rpc ContainerMetrics(ContainerMetricsRequest) returns (↩
   ContainerMetricsResponse) {}
}

message ContainerMetricsResponse {
    float cpu_perc = 1;
  uint64 mem_bytes = 2;
  uint64 disk_bytes = 3;
}

message ContainerMetricsRequest {
    string pod_name = 1;
}
```

The logic of the exposed interface is quite simple:

1. a *client*, usually a metrics aggregator, provides the `Pod name` of a CrownLabs container Instance. He is interested in the Application Container-related metrics;
2. on the basis of the information the metrics scraper saved, the InstMetrics server can respond with:
   (a) current CPU percentage, memory, and disk usage if metrics related to the Pod are found in cache;
   (b) an error if the Pod name is not known to the metrics scraper;

For the sake of simplicity, the provided CPU percentage is relative to a CPU unit. The client will need to normalize the value given the actual CPU units available to the Application Container of the Instance.

4.1.6 Instmetrics daemon architecture

One single Instmetrics scraper is in charge of the collection of all Containerized Instances metrics in a Kubernetes node with very little use of resources. That

component is highly efficient, so it has been decided to run only one Instmetrics per Node.

The service has been deployed as a Kubernetes *DaemonSet*, which ensures that all Nodes run a copy of the Instmetrics Pod, and as nodes are added to the cluster, a new Pod is added to them.

Communicating with the daemon Pod

The first idea was to expose the server on the `NodeIP` and a known port. This approach facilitates traffic routing from clients to the server. Since each Instmetrics only knows about metrics of Instances running on its same Node, it is important for the client to contact the Instmetrics server on its same host. Using the NodeIP, the client only needs to know the host machine IP to reach the correct server.

This approach, however, does not comply with Kubernetes and CronwLabs guidelines, which state to abstract applications' exposition using a *Service* resource. When a Service is used, each Instmetrics Pod automatically gets its own IP address and a DNS name, and the Kubernetes networking plugin takes care of routing.

The service type *ClusterIP* has been used for security reasons: the Service is exposed on an internal IP in the cluster. This makes the Instmetrics server only reachable from within the cluster.

Yet, to meet the requirement of routing traffic inside the Node, an *Internal traffic policy* has been used. Setting the traffic policy as `Local`, all traffic destined for the Instmetrics ClusterIP is only routed to ready node-local endpoints. If no ready server is found, traffic is dropped.

4.2 CrownLabs container Instances infrastructure

This section describes some implementation details of the CrownLabs container-based Instances. Instead, Section 4.3 and Section 4.4 will show how some of the existing components in this area have been extended to support monitoring

and observability.

4.2.1 Sidecar container infrastructure

A CrownLabs container-based Instance consists of a Pod running three different *sidecar* containers.

Sidecar containers pattern consists of a Pod running a main container supported by other containerized processes called sidecars. This pattern allows the extension and enhancement of the functionality of the main container without changing it.

In this case, as depicted in Fig. 4.3, an *Application container* runs along with *Websockify* and a *Display server*.

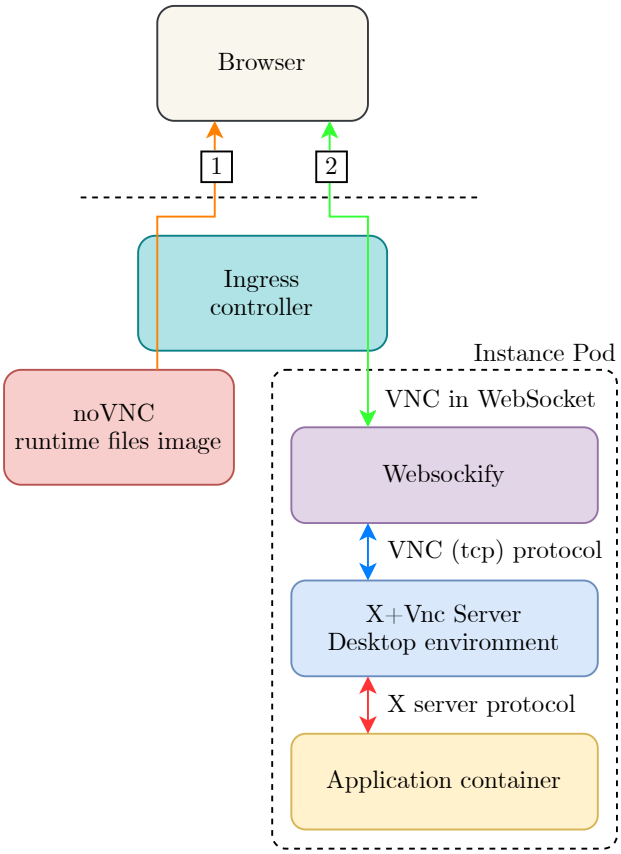

Figure 4.3: Container based Instance infrastructure components

4.2.2 Application container

Application containers are simple processes that have been isolated in a containerized environment. Theoretically, any process can become an Application container simply by writing a self-contained Containerfile[3] for the application layer.

To run an application that features a graphical user interface (GUI), all graphical dependencies must be provided in order to display content. Following the general Docker guidelines, which recommend one concern per container, a sidecar container will be needed to host the display server. In the case of a remote desktop environment, a virtual screen must be used as a display client: this may require an additional sidecar.

4.2.3 Remote display server

Typically, all graphical Linux applications connect to an *X Display server* to show graphical data on the monitor of the computer. The display server is a component of the X Window Sistem (often called *x*): a client/server windowing system for bitmap displays.

The X server is the program that displays the windows and handles input devices such as keyboard and mouse. In the case of the container Instance, the server is needed to elaborate client inputs and send the Application Container graphical output to multiple clients.

Remote display

In order to access the X server from a virtual screen, an interface layer is needed: the application will communicate the data to render to the X server, but remote clients will connect to a *VNC server*.

Xvnc has been used to this end. It includes two servers in one; the X server used by the application to display itself, and a VNC server accessed from remote clients using the VNC protocol. CrownLabs uses TigerVNC to implement this

[3]A Containerfile is a text document containing all the commands a user could call on the command line to assemble an image. A container engine can build images automatically by reading the instructions from a Containerfile.

type of server. TigerVNC strictly fulfills its motive; it can only be used for remote displaying interfaces, but won't work with a physical screen.

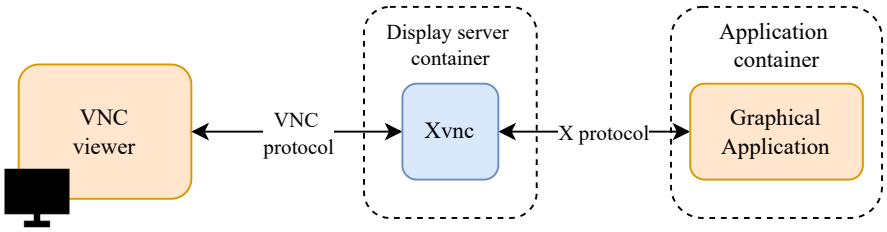

Figure 4.4: Remote display infrastructure

VNC (Virtual Network Computing) is a graphical desktop-sharing system 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. Typically, VNC uses TCP as its transport protocol, so a VNC client is not natively compatible with running within a web browser.

4.2.4 Browser-based remote desktop

CrownLabs provides access to Instances via a web browser, eliminating the necessity for external software to use its remote computing services.

noVNC

In order to enable remote desktop control from a browser, CrownLabs uses *noVNC*: a web implementation of a VNC client. It is both an HTML VNC client JavaScript library and a web application built on top of that library.

noVNC includes a minimal, expandible user interface featuring clipboard sharing and managing of the VNC connection. Thanks to a WebAssebly implementation of the *Remote Frame Buffer*(RFB) protocol[4], it delivers high performances with a low footprint on the client CPU, memory, and network.

[4]Remote Frame Buffer is an open-source protocol for remote access to graphical user interfaces.

noVNC follows the standard VNC protocol, but since browsers do not support native access to TCP sockets, it does require WebSocket support.

WebSocket

WebSocket (RFC 6455) is a standard protocol that enables a web browser or client application and a web server to communicate by using one full-duplex connection layered over TCP. The protocol is designed to work over HTTP ports 80 and 443 as well as to support HTTP proxies and intermediaries.

WebSocket was born to enable browser-based applications with two-way communication without relying on HTTP polling. Alternative solutions that involve existing HTTP technology to accomplish seamless interactive communications are cumbersome and inefficient. Examples are polling or opening two HTTP connections that handle one-way traffic only.

With WebSockets, instead, after an HTTP request-response sequence to establish the connection, data are written and read over one channel only in an asynchronous full-duplex manner. Thanks to the WebSocket JavaScript API, developers are provided with easy tools to send messages to a server and receive event-driven responses.

Websockify

In order to be compatible with noVNC, VNC servers need to use a WebSocket to TCP socket proxy. *Websockify* is a noVNC side-project that provides a simple such proxy.

Websockify accepts the WebSocket handshake, parses it, and then begins forwarding traffic between the client and TCP server in both directions. This tool allows a browser to seamlessly communicate with any remote TCP server simply using the WebSocket API.

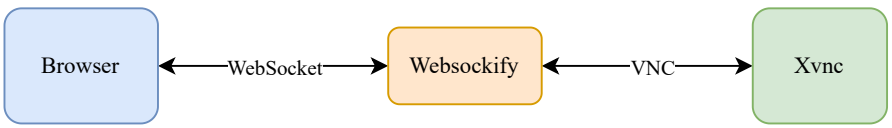

Figure 4.5: Remote-desktop interaction model

In CrownLabs, Websockify runs inside an Instance Pod as a sidecar container, along with the VNC server and the Application container. Configuration is made very easy by sharing the same network namespace with the Xvnc container.

Given the "proxy" role of the Websockify container in the container Instances architecture, Websockify has been effortlessly extended to supply metrics aggregation for the *exams monitoring infrastructure*.

4.3 Metrics aggregation

Many actors can take advantage of CrownLabs **exams monitoring features**. Such features are accessible from a unique, secure interface delivering well-defined structures, regardless of the expected use from the client.¡

4.3.1 Centralized access to Instance metrics

The monitoring infrastructure for CrownLabs exams relies on multiple sources of truth: the *Instmetrics* server provides real-time resources utilization metrics, while *Websockify* holds network information and keeps track of the connections towards the container Instance.

The CrownLabs architecture is designed to be expansible and able to evolve. It is important to not compromise compatibility with metrics clients when the monitoring backend evolves. For these reasons, monitoring capabilities are delivered to clients from a single place hiding the business logic used to scrape metrics.

Websockify has been extended in order to aggregate and proxy access to metrics. The monitoring service is exposed from the same endpoint providing the VNC and noVNC content. While connecting to the remote desktop environment, clients can decide to enable metrics retrieval simply by inserting a parameter in the HTTP request. After the feature expansion, websockify can deliver three different service modes:

- *remote desktop proxy only.* This is the traditional mode, used to retrieve noVNC page HTML content and forward VNC socket data to a websocket;
- *remote desktop and monitoring.* In this case, the client wants to access

the remote desktop environment, and retrieve metrics in order to increase visibility over the Instance he is working on. This mode provides the client also with networking information regarding his connection to the Instance, in order to observe his connection quality;

- *monitoring features only.* This mode is useful for clients interested in observing from the outside the state of the Instance, without exploiting remote computing. A client example is an examiner visualizing the state of an Instance used by a student to carry out an exam. In this case, the client will receive a track of the active and closed connections, besides Application container resource utilization;

4.3.2 Metrics sharing steps

Metrics are shared with web clients using a websocket. This allows pushing resources and connection updates to users, eliminating the need for HTTP polling mechanisms, which turned out to be highly inefficient.

The sequence diagram in Fig. 4.6 depicts the steps to provide the noVNC page with Instance status observability:

1. the user requires access to the remote desktop environment;
2. along with the HTML/JS page content, Websockify includes a `connUID` to use when requesting monitoring for the specific connection;
3. noVNC requires the VNC content and the monitoring feature for the specific `connUID`, indicating how often new metrics must be received;
4. Websockify opens the VNC-related and monitoring websockets;

4.3.3 Tracking Websockify connections

When a connection to the Instance is opened, Websockify identifies the IP address the request is coming from and keeps track of the connection quality measuring the Round Trip Time of a VNC (websocket) packet.

Such measures are stored in the cache in order to be provided as metrics, together with other Instance status information scraped from the Instmetrics.

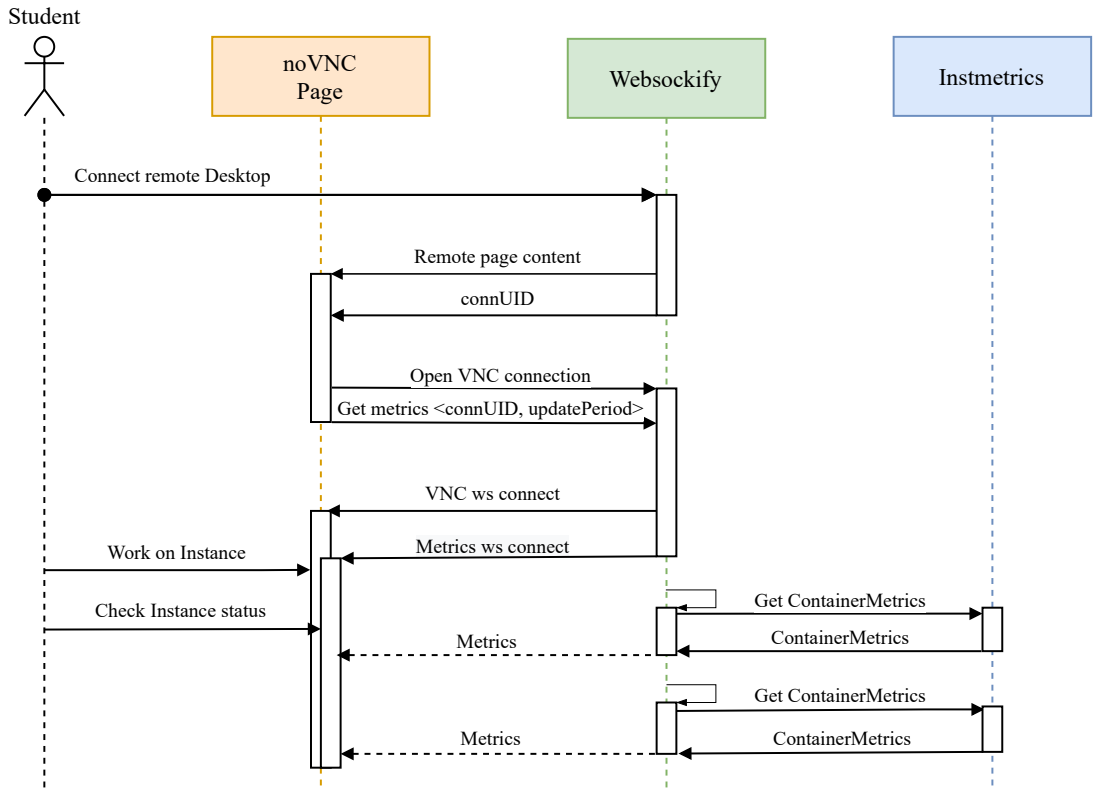

Figure 4.6: Monitoring request steps

X-Forwarded-For Header

The connection origin is carried by the HTTP header *X-Forwarded-For* (XFF): the de-facto standard header for identifying the originating IP address of a client connecting to a web server through a proxy server.

Every HTTP request coming from the Internet and headed to a CrownLabs service, such as Instances, crosses the Kubernetes cluster Ingress. The Ingress runs an Ingress Controller that performs the load balancing of traffic. The CrownLabs ingress controller has been configured to set the XFF header.

It is worth noticing that the request source IP received by the Ingress Controller is often unreliable. For example, it can be modified by intermediary HTTP proxies and, when the Ingress follows a NAT, it can only see the translated IP address.

Measuring Round Trip Time

The Round Trip Time is the amount of time it takes for a ping packet to be sent to the browser inside the VNC websocket channel, plus the amount of time taken by the pong to reach Websockify. Please note that while most of the latency is likely to come from the network, it may be also influenced by client or server overhead. It is, therefore, a reliable indicator of user Quality of Experience.

4.4 Monitoring Frontend

During the design phase of this thesis work, the main actors interacting with CrownLabs exams have been identified and each one has been associated with a set of requirements aiming to increase observability (see Section 3.5.3). Two frontend views have been implemented: a noVNC page **add-on** and a monitoring **dashboard**. The implementation aimed to integrating those new tools into the existing infrastructure as much as possible, in order to enrich users' experience with observability in the most intuitive way.

4.4.1 noVNC monitoring add-on

The noVNC page is designed to deliver a VNC browser-compatible client. The HTML and JavaScript content of the page, however, can be extended to deliver new features to the web page user.

The existing page content has been extended to increase observability over the Instance. Whit the introduction of the monitoring add-on the student is aware of the resource utilization and connection quality, always while interacting with the remote desktop environment. If his remote experience is somehow degraded, he can quickly identify a possible cause and try to recover to an acceptable state for himself.

To reduce the invasiveness of the tool on the page, the Instance health indicator is reduced to a small icon while the student is working on the remote application. The add-on can be moved over the page simply by dragging and dropping it. Fig. 4.7 shows three possible states of the small, semi-opaque, visibility tool, while Fig. 4.8 presents the extended information displayed when

the mouse hovers over the add-on.

Figure 4.7: Small monitoring view examples

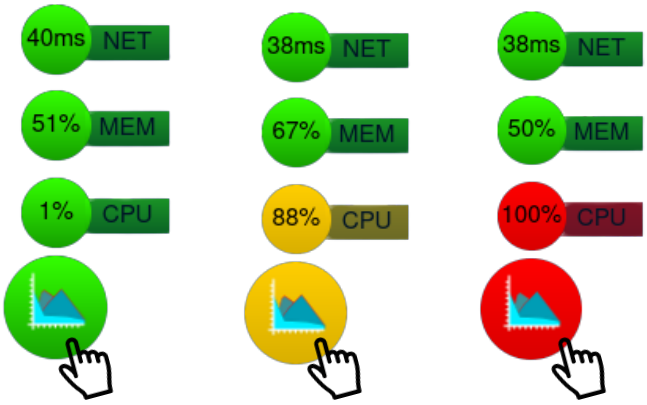

Figure 4.8: Expanded monitoring view examples

Displayed information ar received from a websocket periodically pushing updates:

- the `NET` value refers to the RTT time taken by a packet transmitted on the VNC websocket to go back and forth from the server to the browser;
- them `CPU` and `MEM` values shows the current resource usage percentages on the Application container the student is working on (see Section 4.2.2);

4.4.2 Examiner Dashboard

The second frontend monitoring implementation is a Dashboard providing wider visibility over the Instances involved in a remote exam. This monitoring view is intended to be used by professors issuing a CrownLabs-featured exam on Moodle.

The dashboard is built using ReactJS[5] and, despite being an independent page, it is well-integrated into the existing CrownLabs infrastructure. The web page is hosted on the CrownLabs cluster and it can be reached from a link button on the Moodle exam management page.

In order to provide a summary overview of the exam Instances, the dashboard needs to interact with two CrownLabs components: the examAPI provided by the Exam Agent (see Section 3.4) and the Instance Metrics Aggregator discussed in Section 4.3. Fig. 4.9 shows the interactions between the new dashboard and the existing infrastructure.

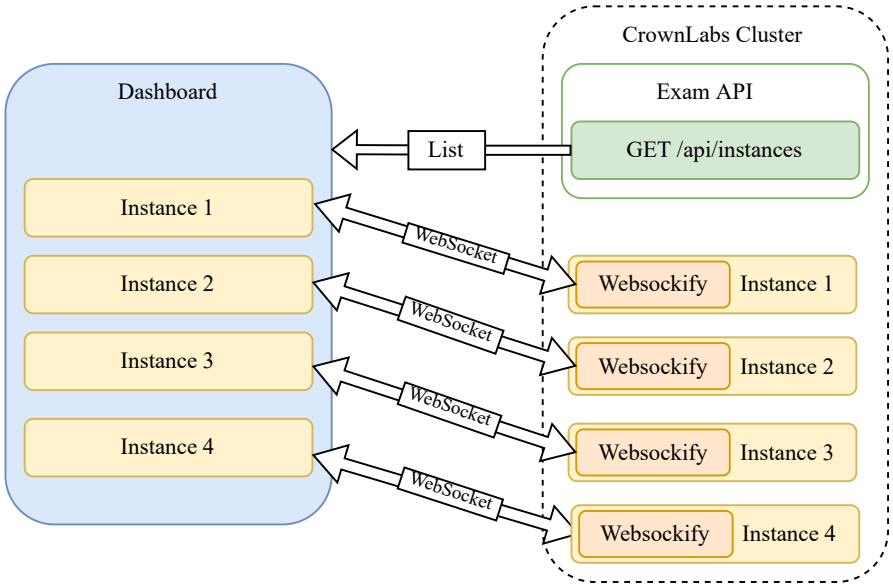

Figure 4.9: Exams Dashboard interactions

Exam Agent API

The CrownLabs examAPI returns the list of existing Instances associated with an exam, creates them, or modifies their state. The Dashboard uses the API to extract the list of Instances associated with a particular exam. Among others, the following useful information is retrieved for each Instance:

[5]React is an open-source frontend JavaScript library used to develop single-page web applications. It is based on dynamic state management representing the desired rendering outcome of the Document Object Model (DOM).

- Instance `ID`;
- whether the Instance is `running` or not. The running state is displayed on the Dashboard to check if the student is still working on the remote environment or if he terminated the quiz;
- the particular `phase` of the running instance. It is useful to check whether the Instance is still starting or is ready for connections;
- `submission` state of the exam project. It will allow examiners to be always aware of which exam terminated with a successful submission;
- `student ID` associated with the remote computing environment;
- Instance `URL`. This allows the dashboard to connect to the *Metrics Aggregator* in order to retrieve monitoring information. The same URL can be used to jump on the remote desktop environment directly from the dashboard;

Metrics retrieval

For each listed Instance, the dashboard will display a monitoring view. Metrics are retrieved from the Metrics aggregator included in the Instance's Websockify component: thanks to a websocket connection, updated metrics are periodically pushed towards the dashboard. Every update will include the following data:

- list of active and past connections to the Instance. The origin IP address is always included, while active connections are associated with their RTT value. These values enrich the network overview provided on the dashboard: examiners will monitor students' connection quality and associate each connection origin to a location.
- current CPU usage of the Instance Application container. This allows observing real-time CPU utilization of the remote environment. In addition to the instantaneous values, the dashboard displays a resource quality indicator computed over past values;
- current memory usage of the Instance Application container;

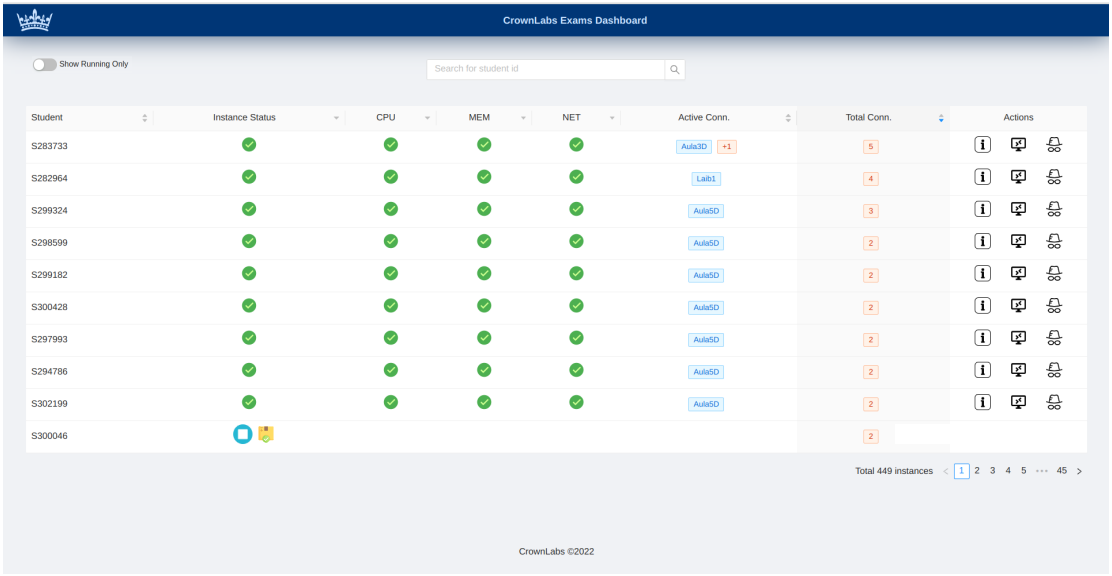

Figure 4.10: Exams Dashboard page

Chapter 5

Validation

The whole software infrastructure, modified to support monitoring features, has been tested upon an existing Kubernetes cluster. Test results in various conditions have been collected to understand the feasibility of the proposed solutions, in terms of resource utilization, performance, and scalability.

After some pre-production measurements, the CrownLabs infrastructure has been used to carry out real university exams, held during July 2022 PoliTO exams session.

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.

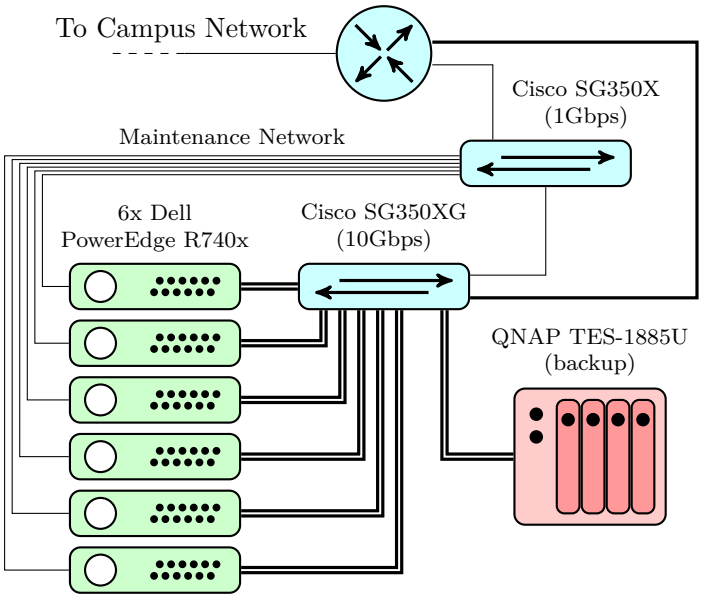

Figure 5.1: Physical infrastructure

The presented infrastructure has been enlarged in a multi-cluster manner. However, most of the tests and operations have been carried out on the main

cluster depicted above. Only some CRI-related validation required using the aggregated cluster.

It is worth noticing that the above cluster's nodes run Kubernetes 1.23 based on *Dockershim*, while the aggregated cluster runs Kubernetes 1.22 with *Containerd* as container runtime.

5.2 Measurements

This section presents some test results, as well as a discussion about differences in performance depending on varying implementations.

5.2.1 Metrics scraping performances

CrownLabs' exams monitoring infrastructure introduces an Instmetrics server collecting resource-related metrics. This component can use both the CRI-API and Docker Engine to scrape container stats (see Section 4.1.1). Which one to use depends on what *container runtime* the Kubernetes node is based on. Both the implementations have been tested, presenting very different results.

While both solutions scale up well and are subject to similar time increase when the number of watched containers raises, the Docker Engine response performances are bounded-below by a time of `1s`. This limitation is inevitable, since the Docker metrics scraping logic presents a hard-coded process sleep lasting 1s.

To test scalability and performances of the two solutions, 55 Instances have been started on the same node. The test has been repeated on a *containerd* and *dockershim* node. Fig. 5.2 and Fig. 5.3 show the respective results.

5.2.2 Production results

The infrastructure has been used to carry out the Computer Science exam. The total number of students booked for the exam session was around 1000, and 938 terminated Instances have been counted at the end of the day.

Given the lack of big-enough classrooms, and to avoid an untested overload

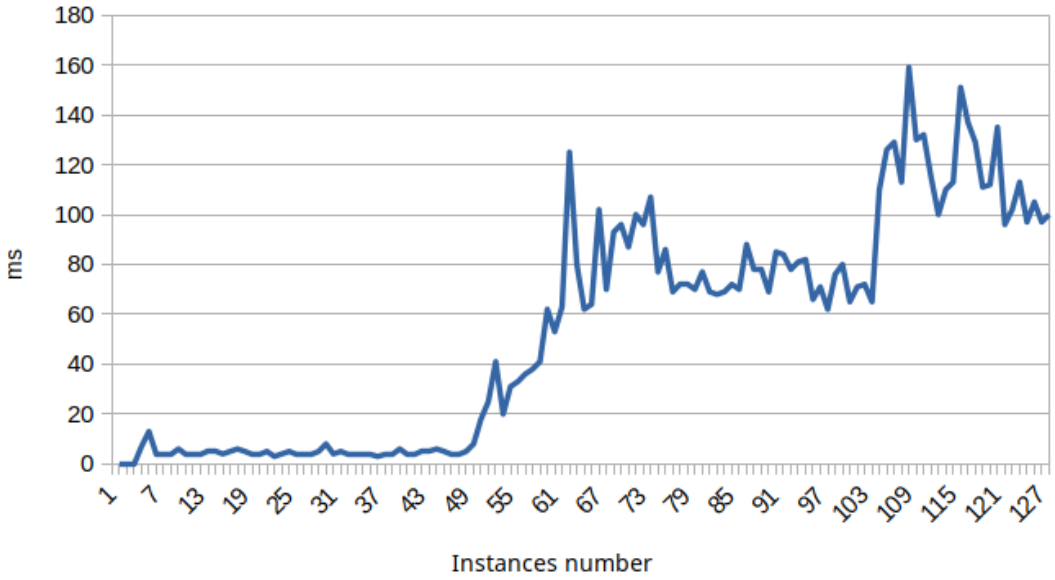

Figure 5.2: Containerd-based CRI-API scraping time

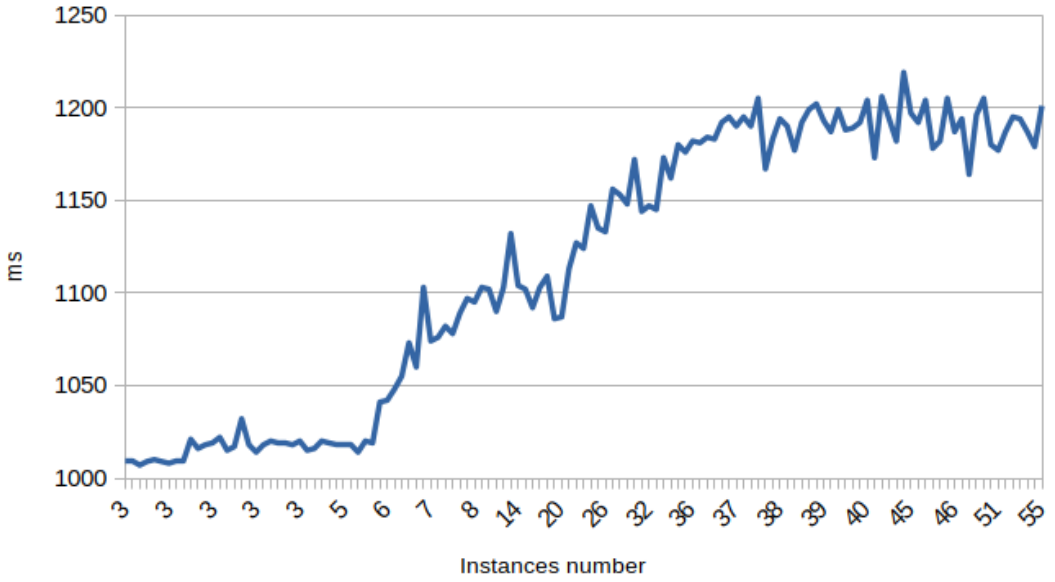

Figure 5.3: Docker Engine API scraping time

of the system, the exam has been divided into 4 rounds. Therefore, the multi-cluster experienced a maximum of 250 Instances active at the same time.

Please notice that the analyzed data refers only to the main cluster presented in Section 5.1, running on average 150 Instances per exam turn.

During the exam day, all other CrownLabs users' authorizations to create personal instances have been disabled.

The resources limit for each Application container has been set as follows:

- CPU limit to 1 virtual core, with 0.75 virtual cores reservation base;
- Memory limit to 2 Gigabytes, with the same 2GB memory reservation;

Instmetrics CPU impact

Fig. 5.4 shows the overall CPU usage over the cluster in terms of threads number: as expected, the main overhead comes from running Instances. It is clear that the start of each exam round characterizes a load peak, since PyCharm, the Application container used for the exam, performs intensive I/O operations during startup, like indexing the project code.

From the Instmetrics processes perspective, Fig. 5.5 shows how each exam round matches with a CPU usage increase, roughly twice the usual usage when few Instance metrics are scraped. However, the CPU impact of the Instmetrics component is negligible compared to Instances load.

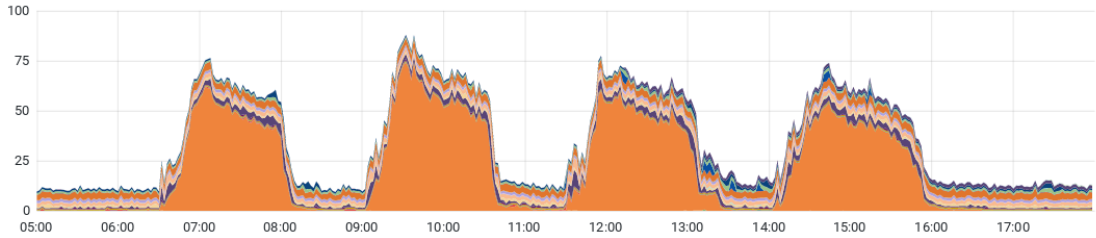

Figure 5.4: Overall CPU usage in July exam session

Focusing on some Instmetrics numbers: the `worker-6` metrics process' went from 0.003 to a peak of 0.024 virtual CPUs usage, watching respectively 3 and 37 Instances.

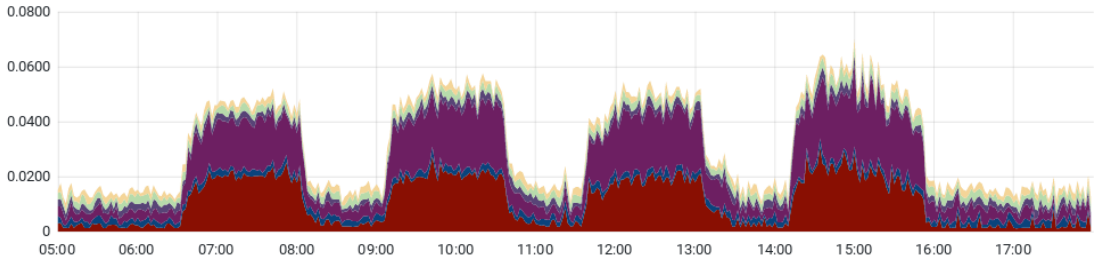

Figure 5.5: Instmetrics CPU usage on July exam session

5.2.3 Instmetrics Memory impact

Also in terms of memory, the Instmetrics daemon had little impact on the overall cluster memory usage. Fig. 5.6 and Fig. 5.7 show respectively the total memory usage and the amount used by Instmetrics.

It is interesting to note how not all the allocated memory is released at exam end from the Instmetrics. Although the scraper clears the cache for un-watched containers, this operation may be delayed by the Go garbage collector, since only 15% of the 150Mib requested memory is used.

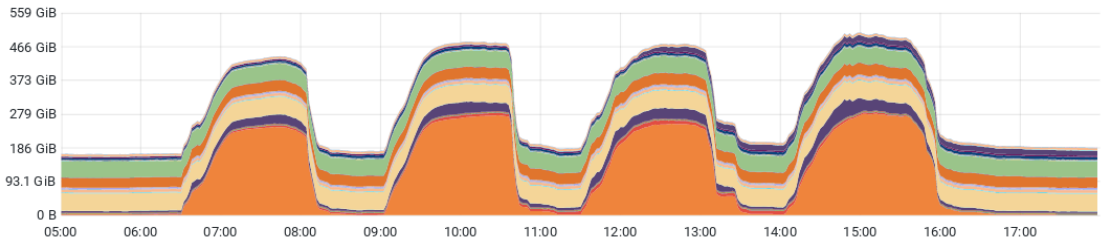

Figure 5.6: Overall memory usage in July exam session

5.2.4 Network usage

An Instmetrics process under load receives on average 3.5kB/s and sends 2.7kB/s. The metrics scraper component receives highly detailed information from the CRI-API; data is then filtered and stored on the cache. This is the reason why the received data flow is higher than the transmitted one.

The network usage, in this case, is comparable with an Instance, which receives 10kB/s but sends on average 200kB/s. The high transmission bandwidth

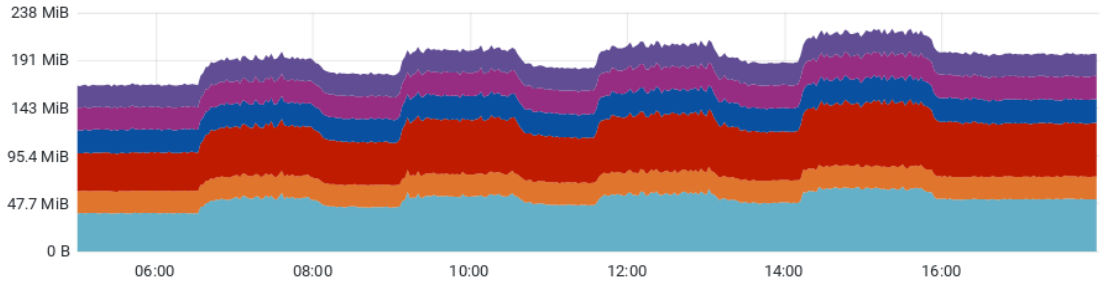

Figure 5.7: Instmetrics memory usage on July exam session

of the Instance is due to the VNC data for the remote display.

Fig. 5.6 and Fig. 5.7 show respectively the total receive/transmit badwidth of Instances and Instmetrics processes.

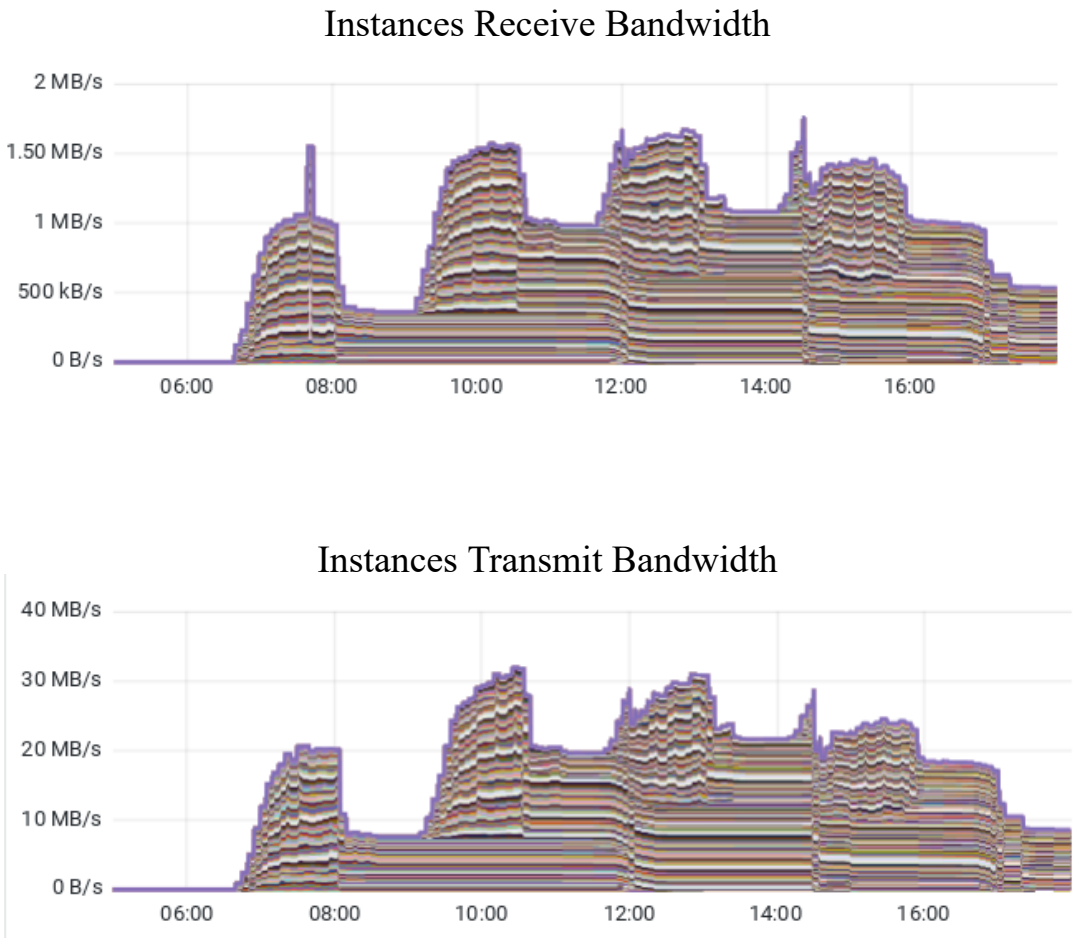

Figure 5.8: Instances bandwidth usage on July exam session

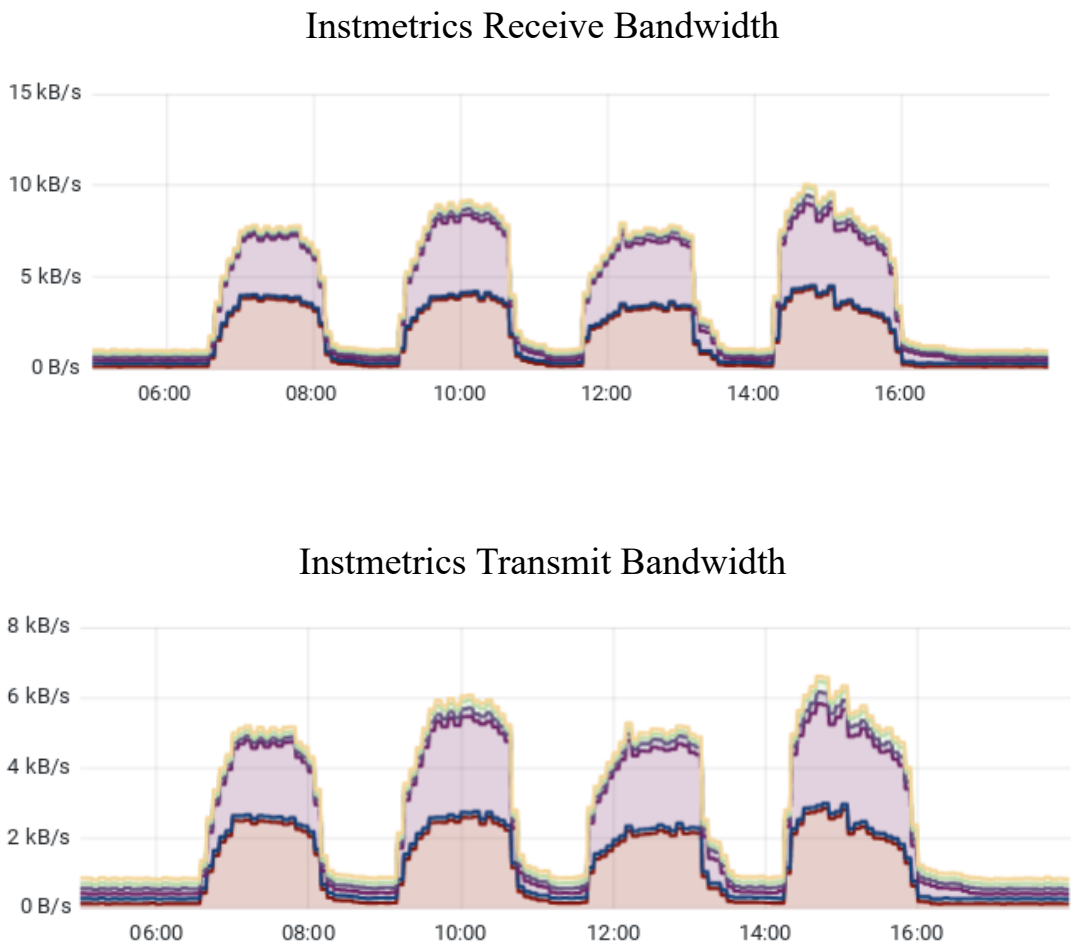

Figure 5.9: Instmetrics bandwidth usage on July exam session

Chapter 6

Conclusions

After the production deployment, feedbacks coming from both students and examiners have been collected. Students made non-intensive usage of the remote desktop environment add-on while being focused on the exam. They appreciated the non-invasive nature of the tool and the decision to expose few but concrete metrics. Examiners, on the other hand, extensively used the monitoring dashboard. For the first time, they had a complete view over the exam Instances, being able to catch suspicious behaviors thanks to the provided resources and connection warnings.

Based on the provided feedbacks, finetuning work has been planned for the dashboard and regarding the provided metrics and their format. Metrics-scraping procedures have proven to be fast, lightweight, and resilient also under high loads. The client-server architecture designed to distribute instance metrics, instead, can be increased in efficiency with the introduction of a centralized broker distributing information for multiple instances.

Given the effectiveness of the extension, the solution has been confirmed to be used in future exam sessions.

In addition to its original intent, this thesis work has also proven the flexibility of modern cloud-native infrastructures. Multiple components have been inserted in an existing production-grade project like CrownLabs with little effort and without compromising the overall architecture.