COMPUTER ENGINEERING DEPARTMENT
LA SALLE UNIVERSITAT RAMON LLULL

Graduation Thesis

Heterogeneous Computing Farm

by

Jiahui Chen

Supervisor: **Dr. Joan Navarro Martín**
Co-Supervisor: **Dra. Anna Carreras Coch**

May 31st, 2022

Abstract

Heterogeneous Computing Farm

The popularity of cloud services is growing exponentially, on the one hand large technology multinationals are investing millions in improving their infrastructure and services. On the other hand, users are increasingly dependent on these services. However, we must not forget that today we have at our disposal, devices for everyday use that exceed the performance of computers that brought humans into space during the last century. These devices are part of the periphery, the edge layer, and are often diverse in nature. We want to make use of this computing power that would otherwise be wasted.The novelty results are twofold: an open-source orchestrator that handles heterogeneous computing agents; an open-source front-end application to interact with these.

Cloud computing, Edge computing, Heterogeneous computing, Distributed systems, Orchestration

La popularitat dels serveis al núvol està incrementant de forma exponencial, per un costat les grans multinacionals tecnològiques inverteixen milions en millorar les seves infraestructures i serveis. Per l'altre costat, els usuaris cada cop depenem més d'aquests serveis. No obstant, no ens hem d'oblidar que avui en dia tenim al nostre abast, dispositius d'ús quotidià que superen en rendiment als ordinadors que van portar l'ésser humà a l'espai durant el segle passat. Aquests dispositius formen part de la perifèria, la capa de l'*edge* i sovint tenen diversa naturalesa. Volem donar-li un ús a aquesta capacitat computacional que altrament estaria desaprofitada. Els resultats de la novetat són dobles: un orquestrador de codi obert que maneja agents informàtics heterogenis; una aplicació frontal de codi obert per interactuar amb aquests.

Informàtica núvol, Computació frontera, Computació heterogènia, Sistemes distribuïts, Orquestració

La popularidad de los servicios en la nube está incrementando de forma exponencial, por un lado las grandes multinacionales tecnológicas invierten millones en mejorar sus infraestructuras y servicios. Por otro lado, los usuarios dependen cada vez más de estos servicios. Sin embargo, no debemos olvidar que hoy en día tenemos a nuestro alcance, dispositivos de uso cotidiano que superan en rendimiento a los ordenadores que llevaron al ser humano al espacio durante el siglo pasado. Estos dispositivos forman parte de la periferia, la capa del *edge* y con frecuencia tienen diversa naturaleza. Queremos darle un uso a esta capacidad computacional que de otro modo estaría desperdiciada. Los resultados novedosos son dos: un orquestador de código abierto que maneja agentes informáticos heterogéneos; una aplicación front-end de código abierto para interactuar con estos.

Computación nube, Computación frontera, Computación heterogénea, Sistemas distribuidos, Orquestración

Contents

List of Figures

List of Tables

List of Code Snippets

Acronyms

- AI: Artificial Intelligence
- API: Application Programming Interface
- ARM: Advanced RISC Machine
- CPU: Central Processing Unit
- EZN: Ephemeral Zookeeper Node
- FPGA: Field Programmable Gate Array
- FS: File System
- GPU: Central Processing Unit
- HTTP: Hypertext Transfer Protocol
- HTTPS: Hypertext Transfer Protocol Secure
- HDD: Hard Drive Disk
- SD: Secure Digital
- SSD: Solid State Disk
- SPOF: Single Point of Failure
- IoT: Internet of Things
- IP: Internet Protocol
- ISA: Instruction Set Architecture
- JS: JavaScript
- JSON: JavaScript Object Notation
- MQTT: Message Queue Telemetry Transport
- MVP: Minimum Viable Product
- NPM: Node Package Manager
- NVME: Non-Volatile Memory Express
- OS: Operating System
- PC: Personal Computer
- PID: Process Identifier

- PU: Processing Unit
- QA: Quality Assurance
- RAM: Random Access Memory
- RPC: Remote Procedure Call
- REST: Representational State Transfer
- SBC: Single Board Computer
- TCP: Transmission Control Protocol
- TPU: Tensor Processing Unit
- LTS: Long-Term Support
- UI: User Interface
- USB: Universal Serial Bus
- UML: Unified Modeling Language
- UUID: Universally Unique Identifier
- VFS: Virtual File System
- VM: Virtual Machine
- ZK: Zookeeper
- ZN: Zookeeper Node

1 Introduction

Executing computational services on the cloud provides certain advantages [Microsoft Corporation n.d.], like scaling, performance and speed. However, it also brings some inconvenience, for instance: (i) additional latency to the sending and reception of the data, (ii) the high energetic consumption, (iii) the downtime of the services which could happen even if the vendor assures a "four nines" (99,99%) uptime availability. There is an alternative to this situation, the usage of edge devices which conforms the edge layer.

1.1 Motivation for Heterogeneous Computing

Traditional homogeneous systems have been decaying at speed of light as opposed to heterogeneous systems, which have been on the rise in the past few years and will prevail. Nowadays, there are a number of compelling reasons as for moving toward heterogeneous environments:

Tailored nodes

The adoption of diversity in cluster makes it possible to have diverse tailored nodes for different computing needs.

Resources usage optimization

Different devices make different resources available. Using these wisely will result in better computation efficiency, better throughput and lower energetic footprint [Mittal and Vetter 2015].

In light of the above, this newer approach explores and exploits its computational capabilities, bringing computation power to a higher limit. However, it also brings some technical difficulties and challenges along, these will be discussed in the next section.

1.2 The complexity of heterogeneity and its challenges

Each heterogeneous device from a distributed system can have different characteristics. Consequently, it can be concluded that for a system to be heterogeneous adds one more dimension of difficulty to deploy an application in it.

1.2.1 Hardware Challenges

Among many hardware challenges [Mittal and Vetter 2015; Zahran 2017], there are a few that should be taken into consideration:

1. Types of Processing Units: CPU, GPU, TPU, etc

2. Processing Unit specific:

    - Architecture of Heterogeneous Computing Systems (Unified or Discrete)
    - Current load on PUs and achieving load balancing between them.
    - Memory bandwidth and PUs data transfer overhead.

3. The memory hierarchy: Usually, the memory hierarchy plays a big role in the performance of a computer system. There is a need to engineer a robust architecture to support heterogeneous cores accesses and optimize the requirements of each. There is also a heterogeneity in these memories since different memories use unalike storage technologies (for caches [SRAM], volatile memory [DRAM], non-volatile memory[Flash, STT-RAM, PCM, etc]) [Meena et al. 2014].

    - Processing Unit's internal storage: Processors' registers & cache. These memories access times [Mistretta and Gault n.d.] are around <1nS to a couple of ten nS, depending on the type. Heterogeneity in memory and data storage types:
        - Registers: <1nS
        - L1 Cache Access: 0,9nS
        - L2 Cache Access: 2,8nS
        - L3 Cache Access: 12,9nS
    - System main storage:
        - RAM memory Access: 120nS
    - System secondary storage:
        - Solid-State Disk: 50 - 150uS
        - Hard Disk: 1 - 10mS

4. Hardware level Interconnection: How should the memory hierarchy and processing cores be connected to squeeze the most performance out. There are a many strategies with regard to topology, such as (Mesh, Torus, Ring, Trees, Star, etc) [Fernandez-Alonso et al. 2012].

1.2.2 Software Challenges

Due to the nature of heterogeneous systems, the machines that compose these systems differ in system architectures, programming models, instruction set architectures. Hence, the software that will run in these machines are build considering the underlying layer.

- Nature of algorithms, for instance, amount of parallelism, presence of branch divergence.
- The underlying Operating System of the host
- Virtualization support

1.3 Objective

The purpose of this project is double-fold. On the one hand a multi-layered distributed architecture will be developed, which will enable the deployment of computational services in various heterogeneous computing nodes either in the cloud or in the edge. On the hand, a middleware will determine the viability of the execution of the software to be deployed, while optimizing the of usage the resources.

Currently the orchestration tools available in the market are backed by big technological companies. Exemplar projects are: Kubernetes [Cloud Native Computing Foundation n.d.] which was originally developed by Google LLC, Amazon's Elastic Container Service [Amazon, Inc. n.d.(a)], Docker Swarm [Docker 2022, May 31], which are container-centric management platforms.

The focus of this project is to develop an orchestration software that could automate these deployments by finding the best node according to the requirements instead of containerizing and virtualizing the applications. To achieve such objectives, the tool should be able to:

- Develop a smart device-agnostic orchestrator.
- Implement service discovery for automated agents connections and disconnections.
- Implement the Application deployment viability algorithm.
- Support for both traditional computing nodes as well as IoT edge devices.
- Embrace heterogeneous environments (devices, OSes, resources).

The second aim of the project is to develop the administrator control panel application with two main ideas:

- An interface to interact with the orchestration tool, API consumption, application deployment upload endpoint
- Display and visualize the orchestrator and cluster's insightful data.

As a result of this project, the Yako (see chapter 2) platform was developed. This open-source project can be found at GitHub [Chen 2022a, May 4; Chen 2022c, May 9].

1.4 Project organization

At the early stages of the senior thesis project a Gantt diagram (see Figure 1) was made to plan and manage the project. The different activities and events with its expected time scale are described in it.

	Task Name	Duration	Start	End
1	Complete TFG Project Execution	255 days	03.11.21	15.07.22
2	Project Planning	16 days	08.11.21	23.11.21
3	Project proposal formal document	11 days	08.11.21	18.11.21
4	Gantt planning	4 days	20.11.21	23.11.21
5	State of the Art (SoA)	22 days	22.11.21	13.12.21
6	Cloud computing examination			
7	Edge computing examination			
8	Infrastructure Evaluation	22 days	13.12.21	03.01.22
9	Equipment layouting			
10	Equipment specification evaluation			
11	Software Requirements Evaluation	22 days	13.12.21	03.01.22
12	Software taxonomy			
13	Software to be deployed evaluation			
14	Orquestration Engine Development	84 days	03.01.22	21.03.22
15	Development			
16	Testing	36 days	21.03.22	25.04.22
17	Software deployment			
18	Analysis	22 days	25.04.22	16.05.22
19	TFG Project Report	183 days	22.11.21	23.05.22
20	Project Defense	1 day	01.07.22	15.07.22

Figure 1: Gantt diagram senior thesis planning

The technical part of the project development was done under the Agile mindset [Agile Manifesto n.d.] with the Scrum [Guides n.d.] and Kanban framework. The different development phases were encompassed into Sprints of 1 week.

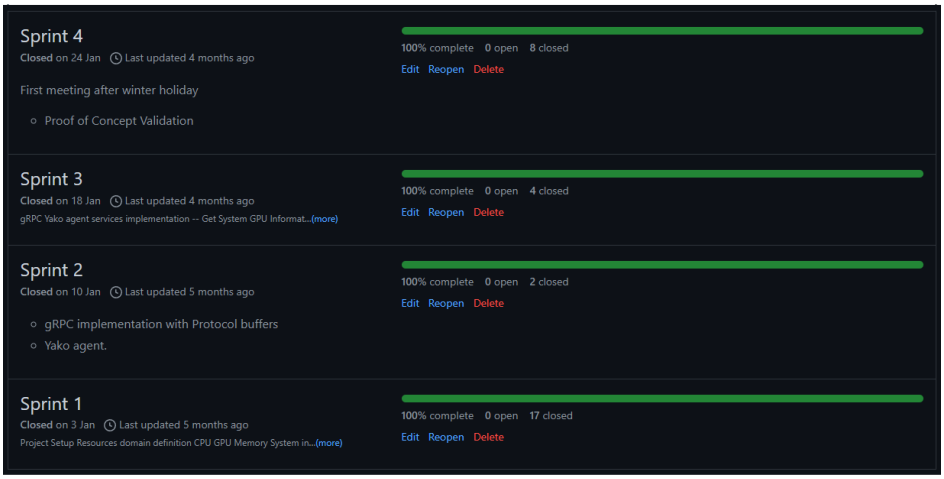
```
|\
| *   69ab732 Merge pull request #41 from JiahuiChen99/feat/40/cluster_controller
| |\
| | * 98fff94 (feat/40/cluster_controller) #40 fix: update service data if exists
| | * 1e934bd #40 feat: register cluster controller to the endpoint
| | * 2e40628 #40 feat: cluster endpoint controller returns cluster schema
| | * 379245a #40 refactor: log newly added service socket
| | * 4a341ae #40 feat: unmarshall newly added service's information and adds it into the correspondent service registry
| | * a0e1915 #40 feat: unmarshall protocol buffers structs into golang models
| | * 01ee918 #40 refactor: removed service registry logs
| | * d66442a #40 refactor: save socket path in the service information struct
| | * fe9911a #40 refactor: services registry now saves service information structs
| | * f4e52bb #40 feat: service information data transfer object
| | * 791391a #40 refactor: register cluster doesn't get zookeeper as argument anymore
| | * 0c3ed50 #40 feat: yakomaster dials service on registration
| | * e31eb71 #40 refactor: removed wait group
| | * 69eabe9 #40 refactor: zookeeper instance is not passed as argument anymore
| | * b90765b #40 refactor: get port as argument
| | * b4ea86d #40 feat: log registered services
| | * 9e84ef4 #40 feat: add new service to the service registry
| | * 434b50c #40 feat: watch all children nodes of the service registry for events
| | * 82132b5 #40 refactor: services registry is now a map
| | * 029188e #40 feat: zookeeper watched events handler
| | * 71a4a2d #40 refactor: zookeeper instance not returned
| | * faa5781 #40 feat: get all service addresses
| | * d02ecd4 #40 feat: service registry list
| | * 3f4ae98 #40 feat: store yakoagent znode uuid for un-registration
| | * 85a13f5 #40 feat: save yako node address when registering the node to the service registry
| | * 1a36f73 #40 feat: yako agent main - Register to service registry - Start gRPC server
| | * b631c7b #40 feat: create zookeeper client instance function
| | * cd4e9f6 #40 refactor: create service registry & register to cluster functions don't create a connection to zookeep
er anymore, it receives as an argument instead
| | * 40f4328 #40 refactor: registry znode path made constant
| | * e556b7d #40 feat: zookeeper register to cluster method
| | * 58ebf92 #40 feat: zookeeper create service registry znode - Creates a permanent service registry in zookeeper
| | * eae591b #40 chore: zookeeper client dependency
| |/
| *   699f285 Merge pull request #39 from JiahuiChen99/38/feat/deploy_controller
| |\
| | * e2df0b8 #38 feat: on app startup check for workdir
| | * 6cf9d34 #38 feat: checks if workdir is available before saving the uploaded file
| | * c402828 #38 feat: directory utils - workDir checks if the working directory is available
| | * 0416a0a #38 refactor: deploy is a post request
| | * 1846e43 #38 feat: save uploaded apps to /usr/yakomaster folder
| | * 9218a8c #38 feat: upload app get uploaded file
| | * b101473 refactor: bad request error message
| | * e6af535 feat: default error builders
| | * 17d2b1e feat: rest error model
| | * 4659a20 #35 feat: add file upload handler to router endpoint
| | * e07392d #35 feat: file upload handler
:|
```

Figure 2: Yako orchestrator back-end development Git log (branches, issues and commits)

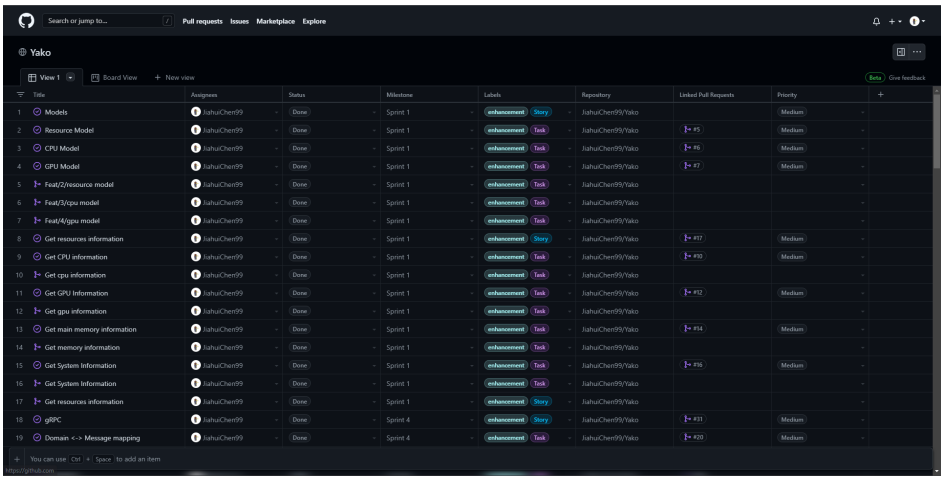
All the developed features were broken down into Stories and further separated into Tech-

nical Tasks, these are known as Issues on GitHub and were assigned with an ID and feature branch. During the development of the project, all the coded features were given its own branch and merged into the development(stable) branch after its implementation. The final release of the MVP is tagged to the main branch. Other best practices techniques, such as atomic commits 2, commits naming [Conventional Commits n.d.], and broad documentation of the software were adopted.

A total of 22 sprints with more than 100 GitHub issues [Chen 2022b, May 4] were produced.



(a) Project Scrum Sprints



(b) Yako project GitHub issues

Figure 3: Scrum and Agile for Yako platform development

2 Yako

The implementation of the project is divided into two parts. On one hand, a back-end, where the orchestrator operates, where logic and computation take place. On the other hand, the front-end, a client that acts as a gateway to interact with all the computational resources available on the server side. A detailed description of all the fractions that comprises the project will be presented in the following sections.

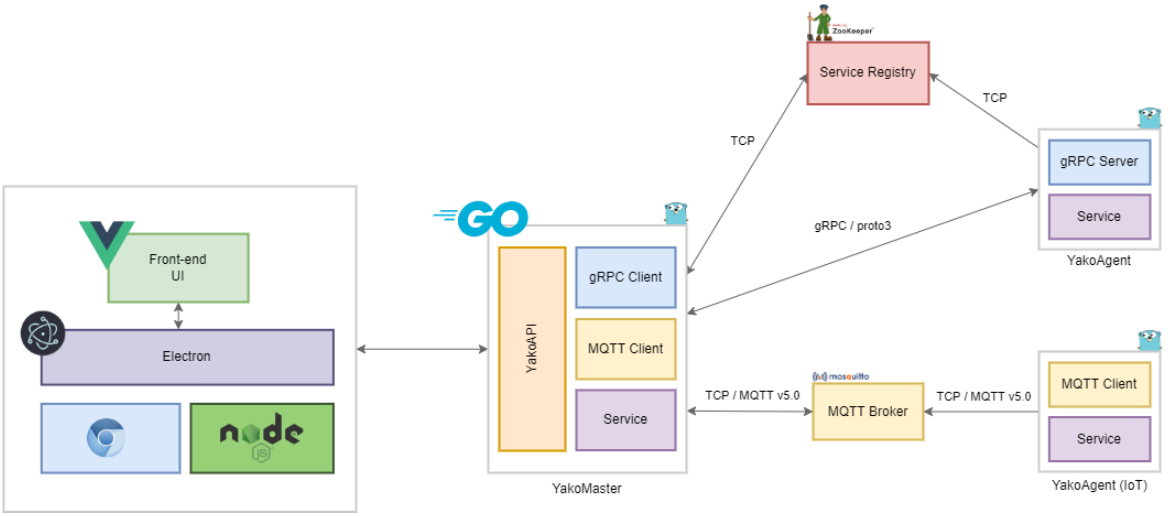


Figure 4: Yako platform simple architecture: (left) front-end application (right) orchestrator and services

Figure 4 represents the internal interconnection of the different components of the platform. An example of a more complex and extended system will be described at the end of this chapter on figure 17.

2.1 Back-end

2.1.1 Architecture

Depending on the specific needs for a cluster, the architecture and internal structure might vary. However, as a general guideline, a computation cluster is generally comprised of computational nodes. As stated in the objective section 1.3, the project aims to address heterogeneous environments, which translates into a context where different types of devices co-exist. Indifferently of the level of heterogeneity these are, in the Yako platform, all devices fall into three main categories which will be introduced in the next subsections. These are the YakoMaster, the YakoAgent and the YakoAgent (IoT).

The platform currently only supports Linux based YakoAgents. Other types of OS are expected to be accepted in the future, this is discussed in the Future work section 5.2.

2.1.2 Service Registry

The service registry, outlined in red in figure 4, is one of the core component of the complete system. It leverages this feature from Apache Zookeeper's distributed system coordination and metadata storage. It is a widely used pattern in real world production environments, like Netflix's infrastructure [Netflix Technology Blog 2017, April 18], Kubernetes [Cloud Native Computing Foundation n.d.], to name a few. The service registry is best suited for environments where the number of services instances and their locations changes dynamically, for instance, usually containers and virtual machines are assigned with dynamic IP addresses. It could also be beneficial for auto-scaling clouds services - number of instances adjustment based on the load.

Originally, ZK was a sub-project of Hadoop and evolved to be a world-class project of Apache Software Foundation. Currently it is adopted by most of the Apache projects, including Hadoop, Kafka, Solr and much more [Reed and Kalmár 2019, September 16]. I decided to opt for ZK in this project because it met both requirements of being an industry leading standard and fit the context of the use case.

The following two paragraphs describe some top features that ZK provides. Further explanation of the usage and implementation of these in the Yako system will be presented in their respective block.

ZNodes

ZK utilises a tree data structure for data storage, each node is called ZNode (ZN). These are named based on the path from the root node. Figure 5 describes how these nodes are represented in memory, further description of the structure pattern used for the project is described in [2.1.3, 2.1.4].

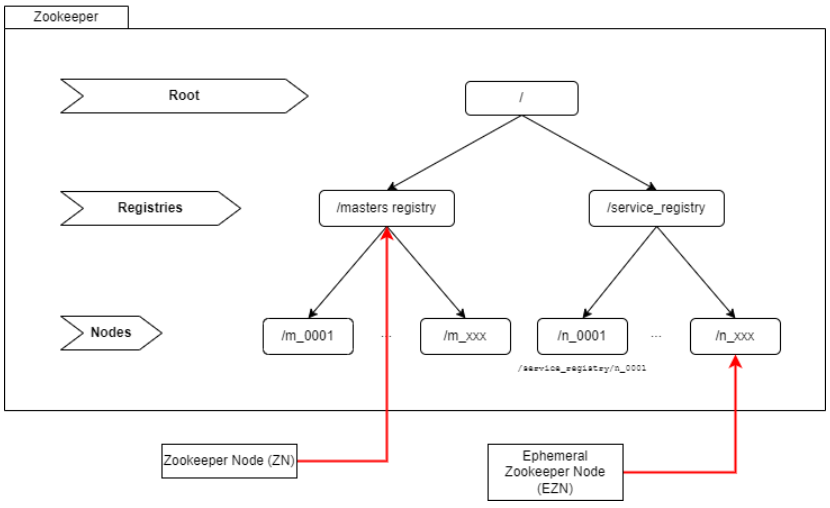


Figure 5: Zookeeper ZNodes structure

Watchers

A observer can be given to any ZN. This watcher observes for any change in the given ZN as shown in figure 6, for instance nodes creation, deletion, data change and finally, addition or removal of child ZNodes. The watch API will notify the listener and this could be handled according to the requirements. The main drawback of ZK's watch implementation is that once triggered, the client must place a new one to continue using the feature.

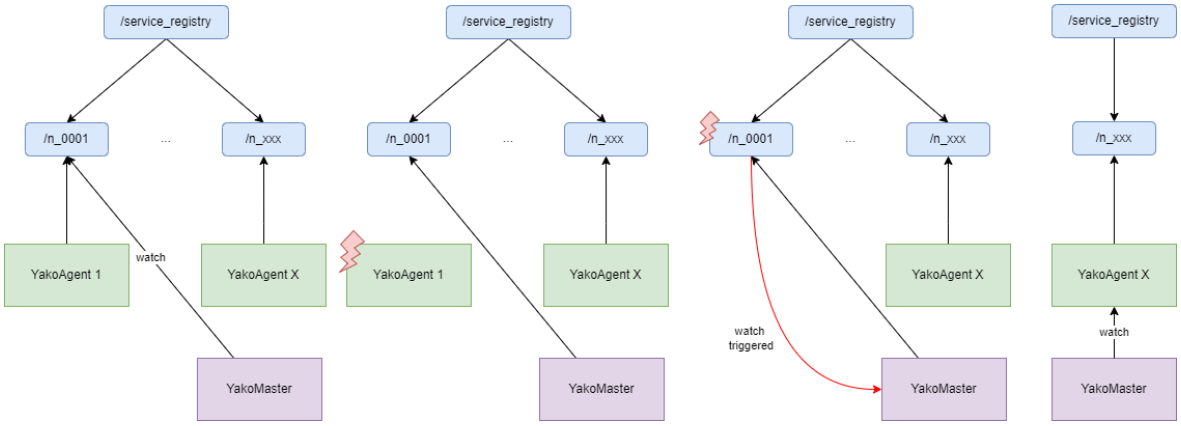


Figure 6: Zookeeper Watchers

2.1.3 YakoMaster

A YakoMaster can considered as the brain of the entire system because it coordinates all the computational resources. There must be at least on of its kind. The following list highlights this server's most important tasks:

- Serves and exposes an API for the client to interact with the system
- Keeps track of the current state of the platform at every single moment
- Identifies the best node to deploy the application according to the requirements
- Handles application upload and deployment

It supports two types of computing apparatus, conventional YakoAgent [2.1.4] and YakoAgent (IoT) [2.1.5] for Internet of Things devices. To enable support for both categories of devices, two standards have been used, gRPC communication for common devices and MQTT communication for IoT based ones.

2.1.3.1 System sequence diagram explanation

As shown in figure 7, on service start up, the agent tries to connect to ZK. If the connection has been successfully established, a singleton client is created for further interactions with the service registry.

Attempts to create a master registry ZN, this node will be created once and only on the first YakoMaster instance startup because it is not an ephemeral but a persistent one. It then register itself to the masters registry and a Zookeeper Node Universally Unique Identifier (znUUID) is provided back. This registers' sole purpose is to track all the coordinators in the cluster and for security purposes.

After that, a registration of UNIX signals processing handlers are established, this captures and processes SIGINT, SIGKILL, SIGTERM events. On YakoMaster Interrupt / Kill / Terminate, the program will gracefully shutdown all the connections to the ZK Service Registry by de-registering itself from it with the znUUID, shutdown the connection with the Mosquitto MQTT broker, gracefully shutdown the gRPC client to all the gRPC servers, shutdown the API server and finally close its running process.

The next procedures are to register to the Mosquitto MQTT broker, serves the API and finally, it stops at an event loop awaiting for new services event.

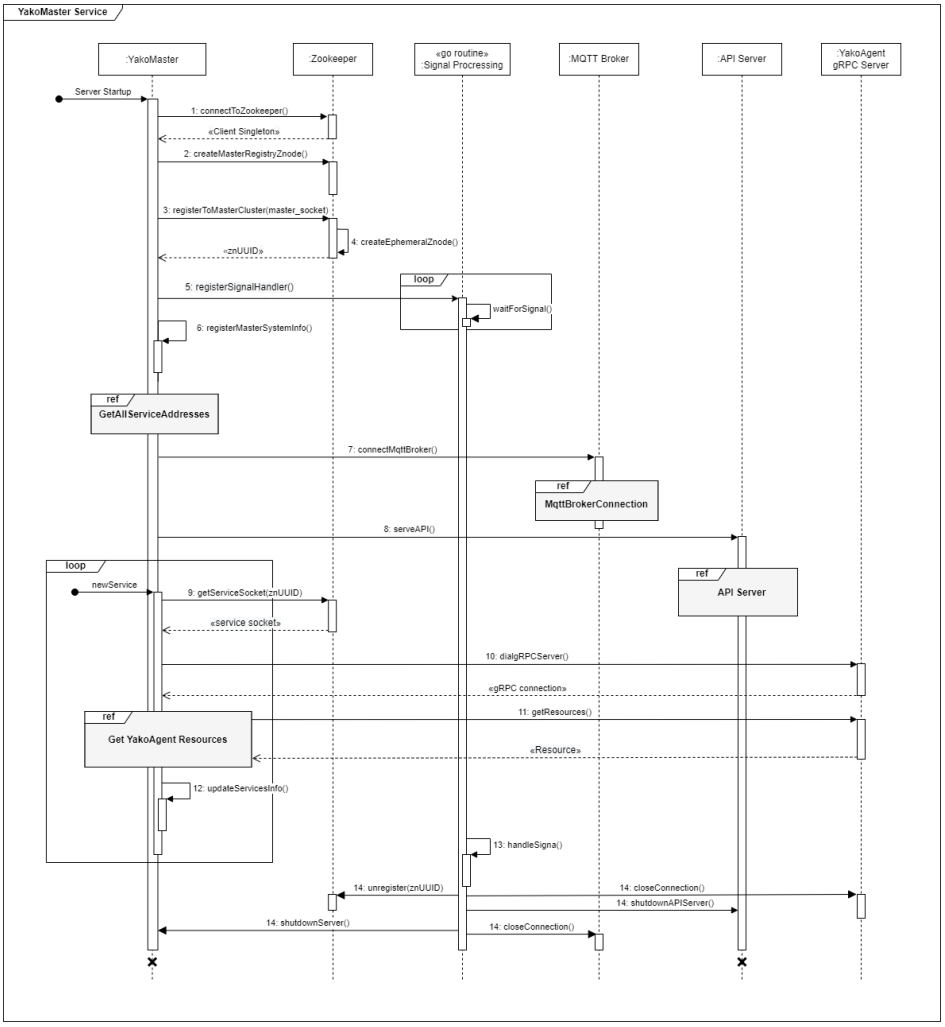


Figure 7: YakoMaster service sequence diagram

Between the signal handler registration and the Mosquitto client connection, YakoMaster will create a new Go routine which will get all services addresses as shown in Figure 8. In this thread, the program will get all children from the service registry and ZK will add a watch to the service registry ZN (/service_registry), figure 5. For each obtained children (service in our cluster), the orchestrator will proceed to get its socket, the meta-data information stored in ZK. If the child did not exist previously, it will prepare and allocate memory to store that YakoAgent's information. It then adds an observer to the child for ZK events. Internally, YakoMaster uses a map data structure to efficiently store the available computing YakoAgents and its system information is retrieved back in $O(1)$ time using the znUUID as the key.

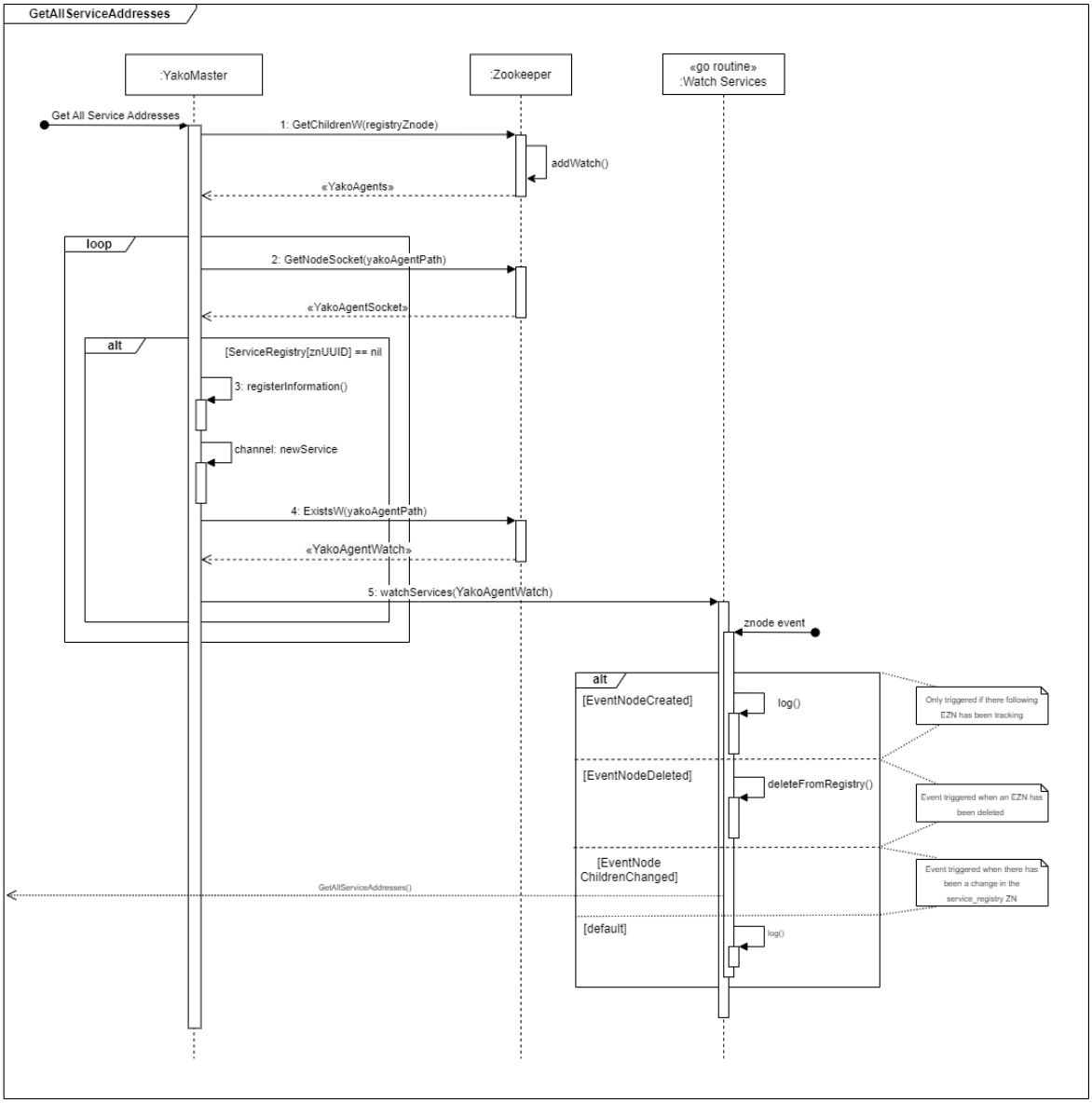


Figure 8: Get all services addresses sequence diagram

The possible ZK events are handled by another Go routine and the event types are listed below:

- **EventNodeCreated** will be triggered on YakoAgent registration only if the following EZN was being tracked.
- **EventNodeChildrenChanged** will be triggered on new YakoAgent registration. It will perform the flow from figure 8, again.
- **EventNodeDeleted** will be triggered on YakoAgent de-registration and disconnection. The handler will de-allocate the memory used in the map.

If the system detects a new YakoAgent, this will unblock the event loop from the main program and will connect to the gRPC server. On connection established it will send a request asking that specific agent to provide its resources information, as illustrated in figure 9. Lastly, the newly obtained information will be updated, also in constant $O(1)$ time, in the internal map.

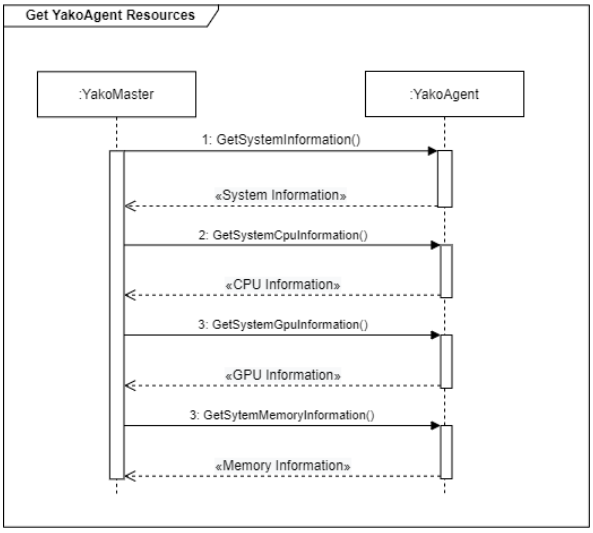


Figure 9: Get YakoAgent resources sequence diagram

2.1.3.2 API Server

A REST API [RedHat 2020, May 8], which is the gateway to interact with orchestrator of the cluster, is served with Gin Gonic Golang framework [Gin Gonic n.d.]. The documentation is exposed at <yakomaster_ip>:<yakomaster_port>/docs and can be accessed with any browser or API testers like curl [cURL n.d.] or Postman [Postman n.d.] / Hoppscotch [Hoppscotch 2022, May 20]. The format complies with the swagger and OpenAPI standard [OpenAPI Initiative n.d.].

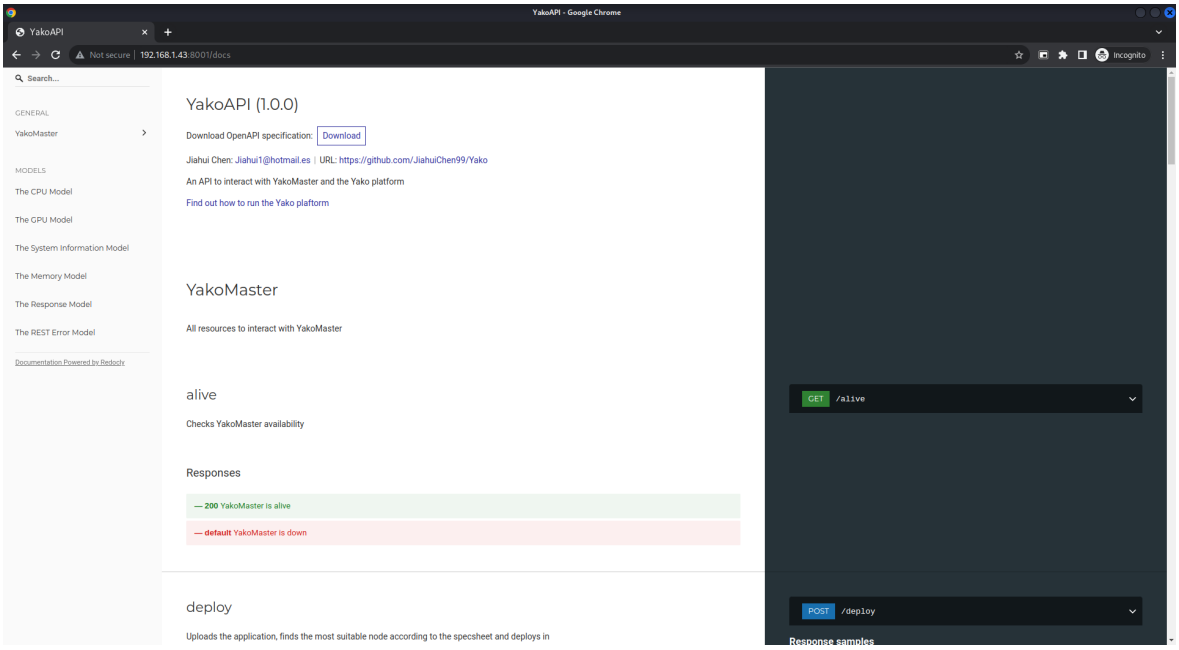


Figure 10: YakoAPI

As depicted in the following figure 11, Postman software has been used to conduct API testing during the project development.

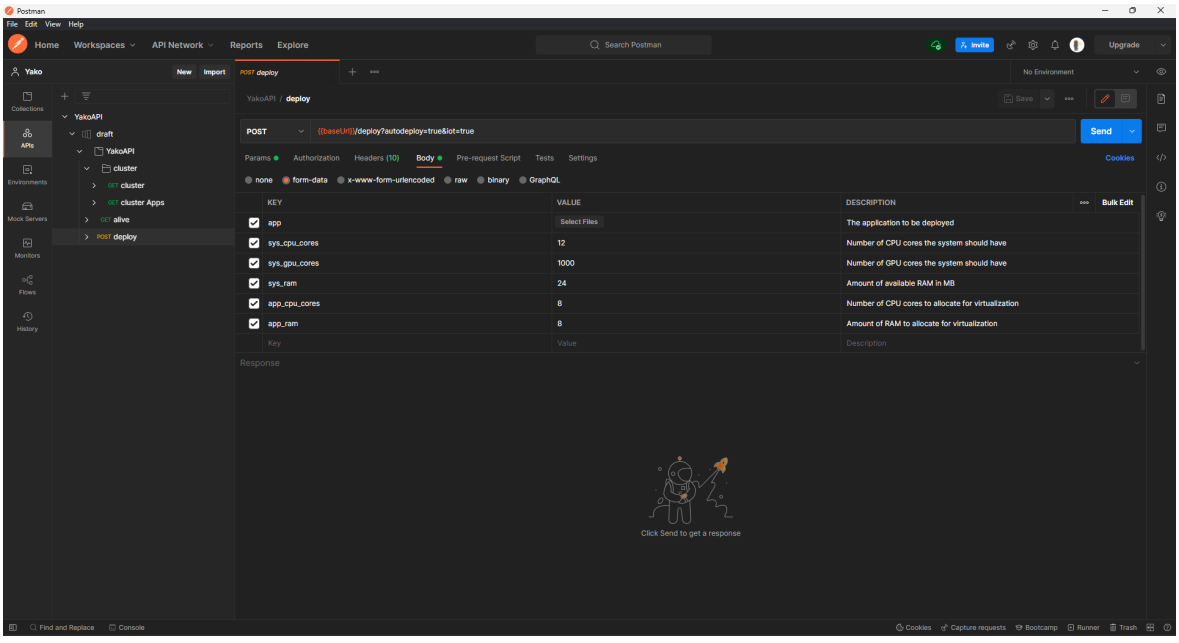


Figure 11: Postman API testing

Code snippet 12a and figure 12b sequence diagram show the API server starting process from the main YakoMaster procedure in figure 7.

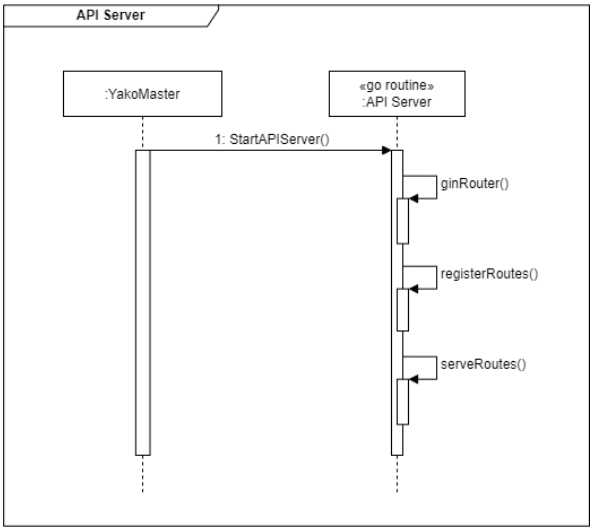
```go
func APIServer() {
        // Default gin router with
        ↪  default middleware:
        // logger & recovery
        router := gin.Default()

        // Add CORS support
        router.Use(cors.Default())

        // Attach routes to gin router
        API.AddRoutes(router)

        // TODO: Use environment
        ↪  variables or secrets
        ↪  managers like Hashicorp
        ↪  Vault
        err := router.Run(addr)
        if err != nil {
                // TODO: Use logger
                panic("API gin Server
                ↪  could not be
                ↪  started!")
        }
}
```

(a) API Server



(b) API Server sequence diagram

Figure 12: API server

2.1.3.2.1 Routes controller

The exposed routes of the API are defined in routing.go file and the following table 1 compiles all the available endpoints and summarizes information related to these routes. Each of the endpoints have an associated controller which will perform business logic.

Table 1: YakoAPI Endpoint

Route endpoint	Method	Controller	Brief Comment
/alive	GET	IsAlive	Pings the server and checks for its status
/deploy	POST	UploadApp	Uploads the application and deploys the it
/cluster	GET	Cluster	Retrieve all the cluster information
/cluster/apps	GET	GetClusterApps	Retrieve all deployed applications

IsAlive

IsAlive is a simple route to check whether the orchestrator is operating or down, it is used before uploading an application in front-end software. The return value of this will be

a HTTP Status OK code: 200 for a, or will timeout HTTP Timeout Code 408 otherwise.

```
package Controller

// IsAlive Handles YakoMaster aliveness
func IsAlive(c *gin.Context) {
        c.JSON(http.StatusOK, map[string]string{"status": "alive"})
}
```

Source Code 1: IsAlive handler - alive.go

UploadApp

As the controller's name suggest, this endpoint is to be requested when the system administrator wants to upload and deploy an application to the system.

```
package Controller

// UploadApp handles the file that the user wants
// to deploy in the cluster
func UploadApp(c *gin.Context) {
        file, formErr := c.FormFile("app")
        if formErr != nil {
                err := utils.BadRequestError(formErr.Error())
                c.JSON(err.Status, err)
                return
        }

        // Check if YakoMaster's working directory is available
        directory_util.WorkDir("yakomaster")

        // Save the file on the server
        appPath := "/usr/yakomaster/" + file.Filename
        if saveErr := c.SaveUploadedFile(file, appPath); saveErr != nil {
                err := utils.InternalServerError(saveErr.Error())
                c.JSON(err.Status, err)
                return
        }

        // Get the app's resources configuration
        var config model.Config
        if err := c.ShouldBind(&config); err != nil {
                c.JSON(http.StatusBadRequest, err.Error())
        }

        var iot bool
        if c.Query("iot") == "true" {
                iot = true
        } else {
                iot = false
        }

```

```
37          // Compute and find the best nodes to deploy the app
38          recommendedNodes := findYakoAgents(config, iot)
39
40          // Check if automation is enabled
41          if autoDeploy := c.Query("autodeploy"); autoDeploy == "true" {
42                  // Auto-deploy the app to the best computed node
43                  yakoAgentID := recommendedNodes[0].ID
44                  log.Println("Autodeploying application to " + yakoAgentID)
45                  if !iot {
46                          // Deploy to a non-IoT agent
47                          deployStatus :=
                             ↪  deployApp(&zookeeper.ServicesRegistry[yakoAgentID]
48                                  .GrpcClient, appPath, file.Filename)
49                          log.Println(deployStatus.Message)
50                  } else {
51                          // Deploy to the best IoT agent
52                          deployAppIoT(yakoAgentID, appPath, file.Filename)
53                  }
54          }
55
56          // File uploaded and stored
57          c.JSON(http.StatusOK, recommendedNodes)
58  }
```

Source Code 2: UploadApp handler - deploy.go

The request's header sent to the UploadApp endpoint must contain a Content-Type [MDN n.d.] property of type "multipart/form-data", this will tell the server the data type of the request. The application to be deployed must be attached to the form in the app field. The client accepts application/json content type back.

Table 2: UploadApp HTTP request headers

Key	Value	Description
Content-type	multipart/form-data	The content of the request i
Accept	application/json	The endpoint accepts JSON formatting

Currently, the supported request parameters are Auto-deploy and IoT. These are compulsory and must be set into the request. As described in the table 3, autodeploy is a variable of boolean type and setting iot to true will only select YakoAgents (IoT), filtering out non IoT devices. Platform and architecture properties are yet to be supported, and will apply OS and CPU architecture (x86-64, ARM, etc [Academic n.d.]) filtering, respectively.

Table 3: UploadApp HTTP request parameters

Key	Support	Type	Description
autodeploy	Supported	Boolean	Automates the application deployment
iot	Supported	Boolean	Applies IoT devices filtering
platform	Unsupported	String	Applies OS filtering
architecture	Unsupported	String	Applies CPU architecture filtering

Table 4 lists all the accepted form keys by the server. Setting these is compulsory for the server to conduct the matching. There are two types of keys: System (sys_xxx) which refers to system requirements, this information will be used to find the most suitable node, and App (app_xxx) used for virtualization which is not supported yet. Other keys could be added in the future to apply finer searching granularity, for instance, latency, disk capacity, disk I/O speeds, RAM speeds, to name a few.

Table 4: UploadApp HTTP request body

Key	Type	Description
app	File	The application to be deployed
sys_cpu_cores	Text	Number of CPU cores the system should have
sys_gpu_cores	Text	Number of GPU cores the system should have
sys_ram	Text	Amount of available RAM in MB
app_cpu_cores	Text	Number of CPU cores to allocate for virtualization
app_ram	Text	Amount of RAM to allocate for virtualization

An example of such request is shown in Code snippet 3, it is a POST petition that accepts form-data as its content type. The selected application is attached to the app key in the form. And the preferred node to run it, should have 12 CPU cores, 1000 GPU cores and 24 GB of RAM, and allocate a total of 8 CPU cores and 8192 MB of RAM for virtualization.

```
curl --location --request POST '/deploy?autodeploy=true&iot=true' \
--header 'Content-Type: multipart/form-data' \
--header 'Accept: application/json' \
--form 'app=@"/path/to/file"' \
--form 'sys_cpu_cores="12"' \
--form 'sys_gpu_cores="1000"' \
--form 'sys_ram="24576"' \
--form 'app_cpu_cores="8"' \
--form 'app_ram="8192"'
```

Source Code 3: UploadApp POST request example with cURL

On request handle, source code 5, YakoMaster will compile the system's and app's requirements from the provided form. Once the information is gathered, the orchestrator will find, by default, the top 3 most suitable nodes. If auto-deploy is enabled, YakoMaster will handle the deployment automatically, otherwise the application will remain uploaded in the system and could be deployed later on. The orchestrator will call the first node and send the application via gRPC streaming upload 2.1.4 if "iot" parameter is not set. Alternatively, if the deployment is to a YakoAgent (IoT), the file would be sent via custom socket streaming to the YakoAgent server as shown in Upload Application reference block from figure 16. Finally, the endpoint will render an array of recommended nodes back to the client, an example is shown in snippet 4.

```
[{
    "ID": "n_xxx",
    "BrowniePoints": 2
},
{
    "ID": "n_xxx",
    "BrowniePoints": 1
}
]
```

Source Code 4: UploadApp POST response example

The heuristic applied to find the best nodes is simple and straightforward. YakoMaster will examine all the available nodes in the platform, firstly by filtering out those that do not match the filtering "iot" parameter. For each of the remaining nodes, the orchestrator will assert if it complies with the properties that the system administrator has set in the form. Currently the system only checks the CPU cores and available RAM. The more it complies the more brownie points will be given to the node. These brownie points will be the determinant factor for the priority queue as a max-heap. In other words, the first node in the heap will be the one with the most brownie points, the YakoAgent that satisfies the most with the requirements. At the current state of the project, with only 2 parameters to check, it is possible that many agents would result in a tie. To solve such event, more parameters are planned to be implemented, as shown previously in table 3. As more properties to check the less likelihood will that happen. To further decrease that unwanted event, a weighting system and/or strict parameters (parameters that must be complied) could be applied to break the deadlock.

```
// findYakoAgents calculates and finds the top X best & suitable
// yakoagents where the app could be deployed according to the
// requested resources from the client
```

```
// Default X = 3
func findYakoAgents(config model.Config, iot bool) []*model.YakoAgent {
        // Priority queue with max heap to rank higher the nodes
        // with more brownie points
        agentsCount := 0
        for agentID, _ := range zookeeper.ServicesRegistry {
                if iot && string(agentID[0]) != "n" {
                        agentsCount++
                } else if !iot && string(agentID[0]) == "n" {
                        agentsCount++
                }
        }
        pq := make(model.PQNodes, agentsCount)

        var browniePoints uint64
        counter := 0
        // Loop through all the available yakoagents, computes the
        // brownie points and adds it to a priority queue
        for agentID, agentInfo := range zookeeper.ServicesRegistry {
                // Filter out the devices depending on the IoT field
                if iot && string(agentID[0]) != "n" {
                        // Set brownie points to 0
                        browniePoints = 0
                        compliesWithCPUCores(agentInfo.ServiceInfo, config,
                        ↪  &browniePoints)
                        compliesWithMemory(agentInfo.ServiceInfo, config,
                        ↪  &browniePoints)
                        pq[counter] = &model.YakoAgent{
                                ID:            agentID,
                                BrowniePoints: browniePoints,
                        }
                        counter++
                } else if !iot && string(agentID[0]) == "n" {
                        // Set brownie points to 0
                        browniePoints = 0
                        compliesWithCPUCores(agentInfo.ServiceInfo, config,
                        ↪  &browniePoints)
                        compliesWithMemory(agentInfo.ServiceInfo, config,
                        ↪  &browniePoints)
                        pq[counter] = &model.YakoAgent{
                                ID:            agentID,
                                BrowniePoints: browniePoints,
                        }
                        counter++
                }
        }
        heap.Init(&pq)

        // Select the top X ones to be recommended
        // X is the number of nodes specified by the user
        x := pq.Len()
        if x > 3 {
                x = 3
        }
```

```
54          recommendedYakoAgents := make([]*model.YakoAgent, x)
55          for i := 0; i < x; i++ {
56                  recommendedYakoAgents[i] = heap.Pop(&pq).(*model.YakoAgent)
57          }
58
59          return recommendedYakoAgents
60  }
```

Source Code 5: Find Top YakoAgents - deploy.go

Cluster

This endpoint will dump all the information recorded in YakoMaster's internal services map. The structure includes all the YakoMasters and YakoAgents. The information will be up to date with at most 10 seconds of age.

```
1   package Controller
2
3   // Cluster returns the cluster schema
4   func Cluster(c *gin.Context) {
5           var response model.Response
6           // Get master & service registries and compose response
7           response.YakoMasters = zookeeper.MasterRegistry
8           response.YakoAgents = zookeeper.ServicesRegistry
9           clusterSchema, err := json.Marshal(response)
10          if err != nil {
11                  log.Println(err)
12          }
13          c.JSON(http.StatusOK, string(clusterSchema))
14  }
```

Source Code 6: Cluster handler - cluster.go

An example of cluster GET request response is shown below, in snippet 7.

```
1   {
2       "yako_masters": {
3         "socket": "127.0.0.1:8001",
4         "sys_info": {
5           "sys_name": "Yako",
6           "node_nme": "YakoAgent",
7           "release": "Linux",
8           "version": "5.18",
9           "machine": "x86_64"
10        },
11        "cpu_list": [
12          {
13            "cpuName": "Intel(R) Core(TM) i7-8550U CPU @ 2.00GHz",
14            "cpuCores": 4,
15            "socket": 0,
16            "cores": [
```

```
            {
              "processor": {
                "value": "0"
              },
              "coreID": {
                "value": "0"
              }
            },
            {
              "processor": {
                "value": "0"
              },
              "coreID": {
                "value": "1"
              }
            }
          ]
        }
      ],
      "gpu_list": [
        {
          "gpuName": "NVIDIA GeForce GTX 1050 Max-Q",
          "gpuID": "0000-1000",
          "busID": "0",
          "IRQ": 0,
          "major": 124,
          "minor": 0
        }
      ],
      "memory": {
        "total": 16384,
        "free": 8908.8,
        "swap": 0,
        "freeSwap": 0
      }
    },
    "yako_agents": {
        ...
    }
  }
```

Source Code 7: Cluster GET response example

GetClusterApps

This endpoint will read YakoMaster's working directory, where all the previously uploaded applications have been stored, and render back the list. This information is used in the Dashboard 25 view, and could allow system administrators to perform an application re-deployment without having to re-upload it.

```go
package Controller

// GetClusterApps returns a list of applications that have
// been uploaded to the system
func GetClusterApps(c *gin.Context) {
        // Read from working directory
        apps, err := ioutil.ReadDir("/usr/yakomaster/")
        if err != nil {
                log.Println("Could not read from working directory
                 ↪  /usr/yakomaster")
        }
        var appsNames []string
        for _, app := range apps {
                appsNames = append(appsNames, app.Name())
        }

        c.JSON(http.StatusOK, appsNames)
}
```

Source Code 8: GetClusterApps handler - cluster.go

An example of the requested applications list is shown in response snippet 9.

```
[
    "yako",
    "super beefy app",
    "executor"
]
```

Source Code 9: Cluster Apps GET response example

2.1.3.3 MQTT broker connection

Figure 13 depicts the process of YakoMaster's connection to the MQTT broker. Eclipse Mosquitto [Eclipse 2018, January 8] has been used for this purpose and uses the publish/-subscribe pattern. The main benefit of this method, is that the orchestrator does not need to repeatedly poll the publishers (YakoAgent IoT) for updates. The IP and port of the broker is provided as programs arguments on startup. A new MQTT client configuration is created with the builder software pattern. With the generated configuration, a connection is then established with the broker. Once connected YakoMaster subscribes to all the available topics. All the available topics a YakoAgent publishes are listed in section 2.1.4.

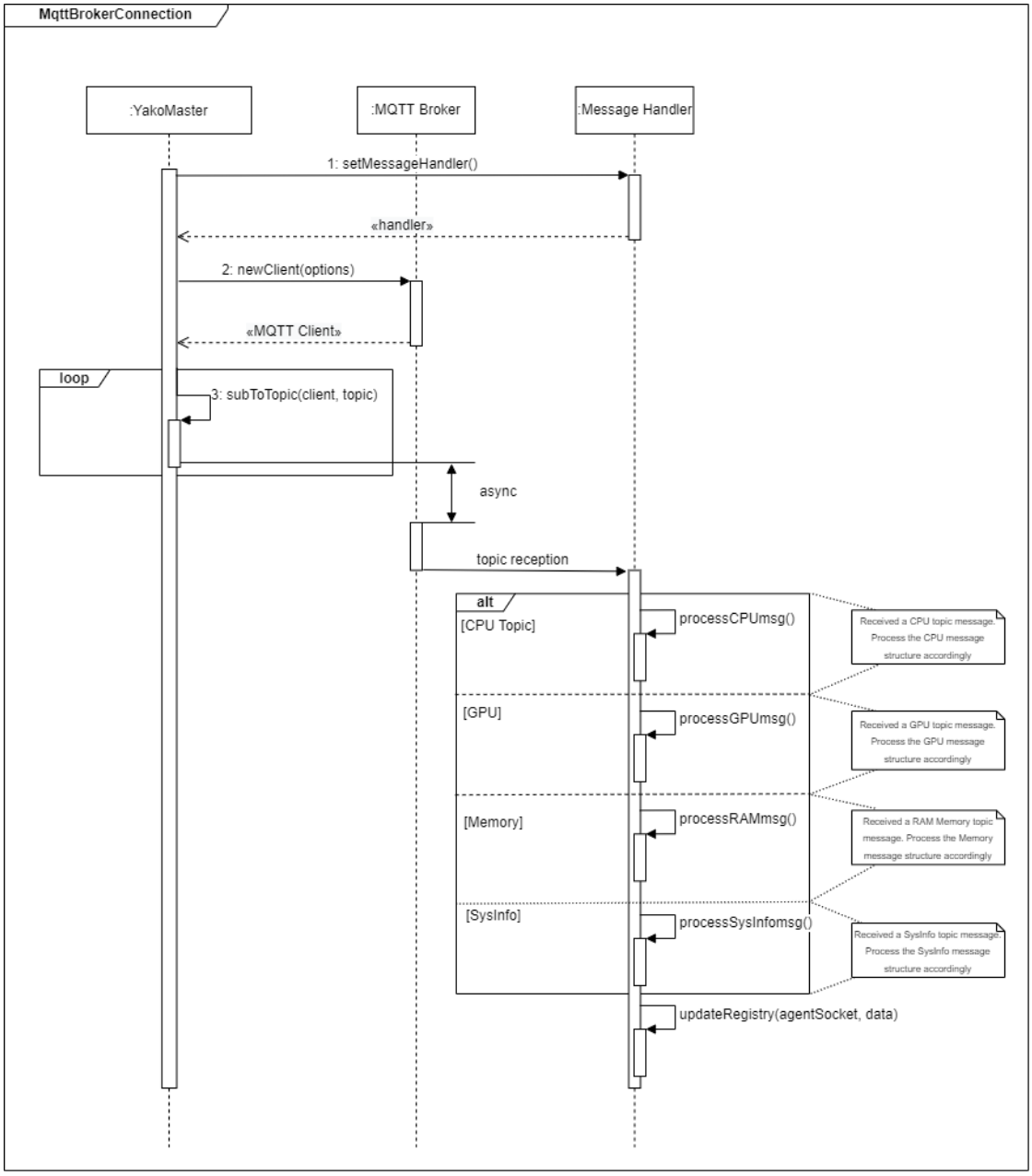


Figure 13: Connect to MQTT Broker sequence diagram

The receiving of the topics are asynchronous and can happen at any time during the runtime. Once a topic is received, a message handler 10 processes it. The messages have the following format:

$$/topic/+/resource$$

The first topic level indicates that it is a topic, succeeded by a single-level wildcard, the + sign, and finally the second topic level indicates the resource that its being received. The wild-card acts as a variable and is substituted by the socket of the publisher. With this information the message handler from YakoMaster can determine the origin of the message and the resource type information that the packet contains. The MQTT published messages payload is serialized in JSON format, but later releases support for protocol buffers could be supported to diminish network usage footprint.

```
package mqtt

const (
        CPU          = "cpu"
        GPU          = "gpu"
        Memory       = "memory"
        SysInfo      = "sysinfo"
        // MQTT Topics
        TopicCpu     = "topic/+/" + CPU
        TopicGpu     = "topic/+/" + GPU
        TopicMemory  = "topic/+/" + Memory
        TopicSysInfo = "topic/+/" + SysInfo
)

// messageHandler Callback handler for subscribed events. Processes the event
// according to the message topic
func messageHandler(client mqtt.Client, msg mqtt.Message) {
        // Topic parsing topic/<agent_socket>/<topic_name>
        mqttTopic := strings.Split(msg.Topic(), "/")
        agentSocket := mqttTopic[1]
        switch mqttTopic[2] {
        case CPU:
                var cpu []model.Cpu
                if err := json.Unmarshal(msg.Payload(), &cpu); err != nil {
                        log.Println("Err", err)
                }
                updateRegistry(agentSocket, cpu)
        case GPU:
                var gpu []model.Gpu
                if err := json.Unmarshal(msg.Payload(), &gpu); err != nil {
                        log.Println("Err", err)
                }
                updateRegistry(agentSocket, gpu)
        case Memory:
                var memory model.Memory
                if err := json.Unmarshal(msg.Payload(), &memory); err != nil {
                        log.Println("Err", err)
                }
                updateRegistry(agentSocket, memory)
        case SysInfo:
                var sysinfo model.SysInfo
                if err := json.Unmarshal(msg.Payload(), &sysinfo); err != nil {
                        log.Println("Err", err)
```

```
                }
                updateRegistry(agentSocket, sysinfo)
        }
}
```

Source Code 10: MQTT message handler

2.1.4 YakoAgent

A YakoAgent is a computational node, it is intended to be a node where applications will be deployed. Internally the YakoAgent starts a gRPC server which implements a service called **NodeService**. The NodeService describes a series of Remote Procedures calls which must be implemented depending on the need. The implementation of these RPC will be system dependant, how the resources are described differs from one Operating System to another. For instance, to get system CPU data, in Linux distributions this information is located in the /proc/cpuinfo file, while in Microsoft Windows OSes, to read that information, a query to Performance Counters [Karl-Bridge-Microsoft n.d.(a)] or Process Status API [Karl-Bridge-Microsoft n.d.(b)] must be made instead.

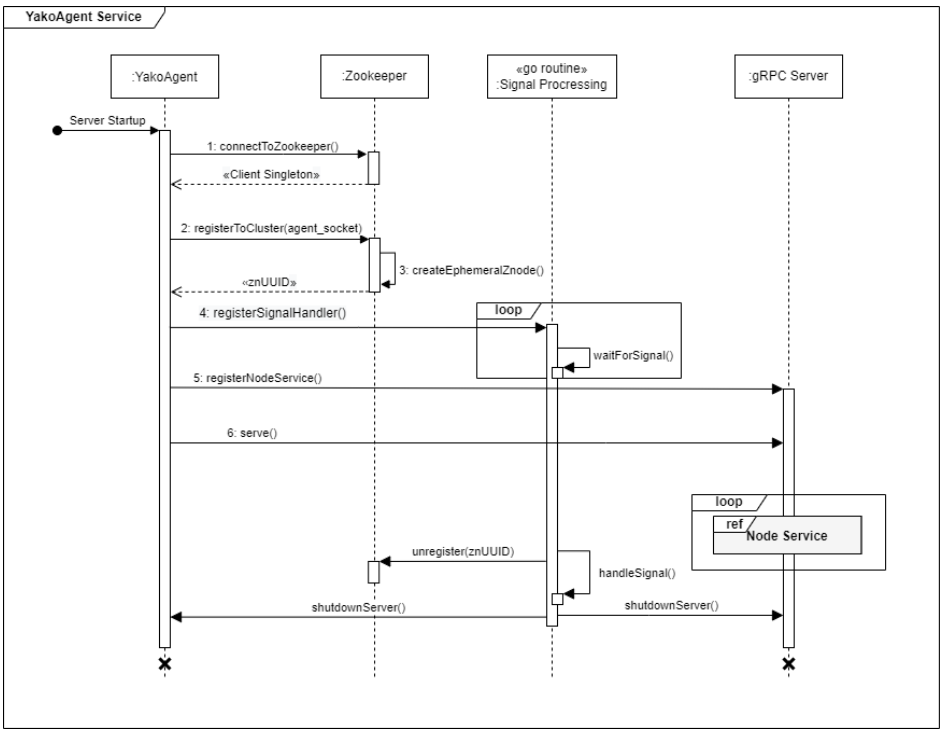


Figure 14: YakoAgent service sequence diagram

Both, service and RPCs are defined in a *.proto* file shared across all the systems that uses gRPC in the Yako platform, YakoMaster and YakoAgent. Most of the information system

resources information are retrieved from the /proc folder, which is a special folder, a Virtual File System (VFS) instead of a physical File System (FS) mounted on boot time.

```
// node.proto defines the services and RPCs a YakoAgent exposes
syntax = "proto3";
// gRPC package
package yako;
import "google/protobuf/empty.proto";
import "cpu.proto";
import "gpu.proto";
import "memory.proto";
import "sysinfo.proto";
import "deploy.proto";

// Go related package
option go_package = "yako/";


// Defines the services that a yako node offers
service NodeService {
  // Retrieves system general information
  rpc GetSystemInformation(google.protobuf.Empty) returns (yako.SysInfo);
  // Retrieves system CPU related information
  rpc GetSystemCpuInformation(google.protobuf.Empty) returns (yako.CpuList);
  // Retrieves system GPU related information
  rpc GetSystemGpuInformation(google.protobuf.Empty) returns (yako.GpuList);
  // Retrieves system main memory related information
  rpc GetSystemMemoryInformation(google.protobuf.Empty) returns (yako.Memory);
  // Transfer & deploy application to the YakoAgent
  rpc DeployAppToAgent(stream Chunk) returns (yako.DeployStatus);
}
```

Source Code 11: YakoAgent services and RPCs API

The use gRPC as the communication protocol for the agents is due to its widely adoption in the cloud, distributed, computing context. gRPC can efficiently connect services across distributed environments. Its ubiquitous adoption, support for multi-platforms and high performance makes it the best candidate.

GetSystemInformation RPC

This RPC will gather generic information of the system where the YakoAgent is running on. Internally it makes a *uname* system call [MacKenzie n.d.]. The OS presumably knows its name, release, version and hardware. Part of this information can be read from /proc/sys/kernel/{ostype, hostname, osrelease, version, domainname}.

```
package model

// SysInfo
// Wrapper struct of 'Utsname' returned by 'uname' system call
```

```
type SysInfo struct {
        SysName  string `json:"sys_name"`
        NodeName string `json:"node_name"`
        Release  string `json:"release"`
        Version  string `json:"version"`
        Machine  string `json:"machine"`
}
```

Source Code 12: System Information model - sysinfo.go

GetSystemCpuInformation RPC

This RPC will gather CPU related information. The information is located in the /proc/cpuinfo path. As shown in the Source Code 13, the most relevant information (name, cores number, socket, list of cores) of the CPU is retained.

```
package model

// Core
// Representation of a CPU core
type Core struct {
        Processor uint64 `json:"processor"` // "processor" field
        CoreID    uint64 `json:"coreID"`    // "core id" field
}

// Cpu
// Representation of a processor modeled after /proc/cpuinfo
type Cpu struct {
        CpuName  string `json:"cpuName"`  // CPU model name
        CpuCores uint64 `json:"cpuCores"` // Number of CPU cores
        Socket   uint64 `json:"socket"`   // "physical id" for multiprocessor
        ↪ systems
        Cores    []Core `json:"cores"`
}
```

Source Code 13: CPU model - cpu.go

GetSystemGpuInformation RPC

This RPC will retrieve GPU related information if any. Not all systems have GPU support, that is the reason why the program first checks for a GPU availability if none a log will be prompted to the terminal. The platform currently supports NVIDIA graphical processing units, and the correct drivers must be installed in order to perform the lookup. If all the previous conditions are met, the folder will be checked. The structure contains the following properties.

```
package model

// Gpu
```

```
// Abstraction of a GPU
// TODO: Get CUDA Cores
type Gpu struct {
        GpuName string `json:"gpuName"` // GPU model name
        GpuID   string `json:"gpuID"`   // "GPU UUID"
        BusID   string `json:"busID"`   // "Bus Location" PCIe bus ID
        IRQ     uint64 `json:"IRQ"`     // "IRQ" GPU Interrupt lane
        Major   uint64 `json:"major"`   //
        Minor   uint64 `json:"minor"`   // "Device Minor" for /dev/nvidia<minor>
        ↪  character device
}
```

Source Code 14: GPU model - gpu.go

GetSystemMemoryInformation RPC

This RPC will report the amount of Random Access Memory (RAM) the system has installed. Information about the consumed and availability of both RAM and Swap virtual memory, as described in Source Code 15.

```
package model

// Memory
// Representation of the main memory, the unit is in kB
type Memory struct {
        Total     uint64 `json:"total"`     // "MemTotal" system installed RAM
        ↪  memory
        Free      uint64 `json:"free"`      // "MemFree" system unused RAM memory
        TotalSwap uint64 `json:"swap"`      // "SwapTotal"
        FreeSwap  uint64 `json:"freeSwap"` // "SwapFree"
}
```

Source Code 15: Memory model - memory.go

DeployAppToAgent RPC

The YakoAgent service follows a life-cycle depicted in Figure 15 during its lifespan. On server start up, before operating, it first connects to Apache Zookeeper to obtain a singleton client, which will be used during the entire session to communicate with the Service Registry utility from Zookeeper. Shortly after the client obtaining, it registers to the Service Registry, this will dispatch an event that will create a sequential Ephemeral Zookeeper Node (EZN). After its completion a unique identifier (znUUID) will be returned back to the agent. This key has various purposes, for instance, node identification, node de-registration, node watcher. The value of each newly generated identifier will be unique and auto-incremented, following n_<auto_increment_ID> pattern. ZK has support for EZNs, these are special ZNodes that only exists as long as the session that created them is active. If the YakoAgent is disconnected either due to a failure or manual shut-down, the session ends, consequently the ZN is deleted.

This is reported back and notified to the YakoMaster which is constantly monitoring the state of the entire system. After that step, the service will register a signal handler to process all incoming signals either from the system administrator (User) or from the OS (External Event), at any time. The Node Service is then registered to the gRPC Server following by a service exposure. To stop YakoAgent from operating, three signals can be used, these are Signal Interrupt (SIGINT), Signal Terminate (SIGTERM) and Signal Kill (SIGKILL).

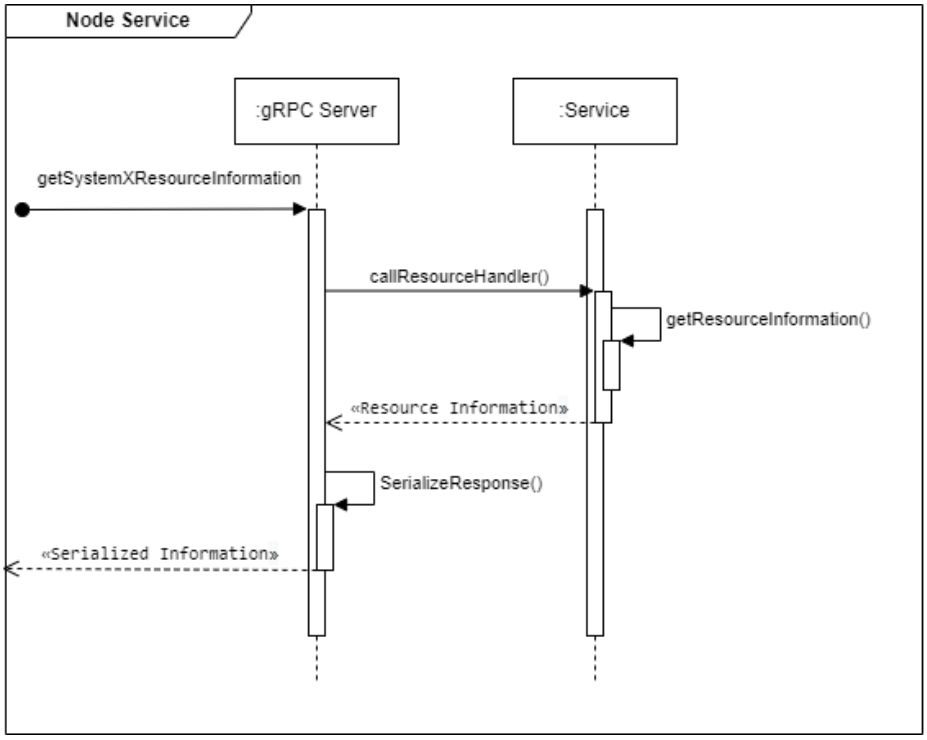


Figure 15: YakoAgent gRPC service sequence diagram

2.1.5 YakoAgent (IoT)

YakoAgent (IoT) is a variant of the previous server, specifically designed to support IoT devices. It replaces the gRPC for MQTT [OASIS n.d.] message queues, which implements the publish/subscribe pattern [Camp n.d.]. As opposed to the standard YakoAgent, this server does not need to connect to the Service Registry to make its appearance in the system. It connects to the MQTT broker and periodically publishes its own system resources, the default timed reporting is 10 seconds but a higher or lower value could be set according to the system preferences.

The following list enumerates the different topics available for YakoAgents (IoT) to self report its resources to the YakoMaster.

- **topic/<agent_socket>/cpu:** To report system's CPU related information

- **topic/<agent_socket>/gpu:** To report system's GPU (if any) information
- **topic/<agent_socket>/memory:** To report system's RAM memory
- **topic/<agent_socket>/sysinfo:** To report system generic information

YakoMaster will be notified once there are information of any of the previous topics. It first performs a check and determines if the node exists in the service registry. If it was aware of it then it updates, otherwise it creates a new registry.

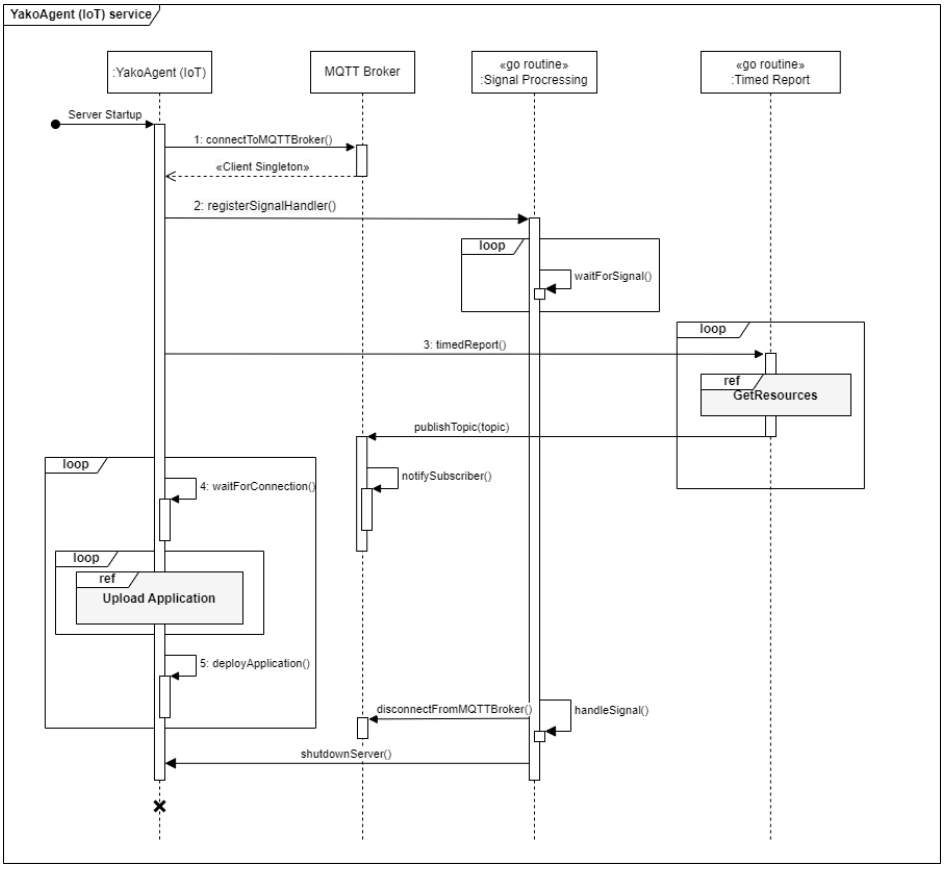


Figure 16: YakoAgent IoT service sequence diagram

2.1.6 A more complex system

One of the core objectives for this platform to achieve, was to be a language agnostic, to work across platforms embracing the heterogeneity nature of the devices, hardware and software wise.

A more complex system is presented in the following figure 17. Both YakoAgent and YakoAgent (IoT) clients could be developed with different languages according to the specific needs of the device.

It is known that the majority of IoT devices that constitute the edge layer of a distributed system are more resource constrained [Rajaee 2017]. That is why system languages like Rust, C++ and Go with less memory consumption are more suitable, YakoAgent (IoT) clients for these language could be developed. For these devices to publish topics to the MQTT broker, a MQTT library of that language must be used, which all the aforementioned languages do have.

On the other hand, for YakoAgent nodes it is as simple as getting the different protocol buffers files and compiling for the programming language being used. Yako platform will support this software layer heterogeneity thanks to its unified messages protocols, gRPC with protocol buffers encoding and MQTT with JSON-encoding.

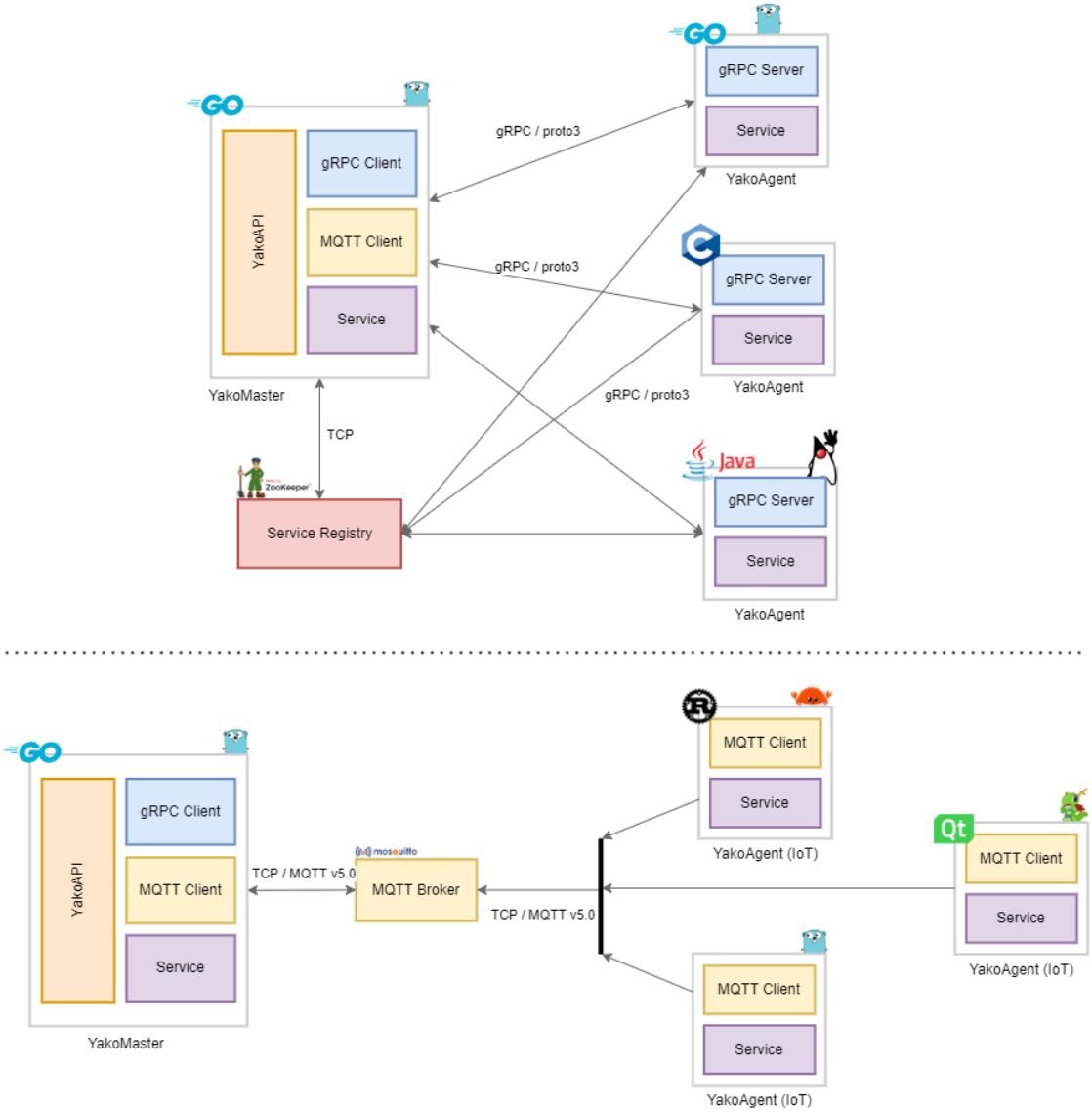


Figure 17: Yako platform complex architecture (more heterogeneous): (top) other YakoAgent (down): other YakoAgent (IoT)

2.2 Front-end

The front-end application is the control panel of the entire Yako platform. One of the key considerations when designing the application was, for this to be available to be used from any device. The software has been developed with the latest web technologies. It is a multi-platform application, hence, the software can be ran in diverse OS and supports mainstream platforms Microsoft Windows, Linux, Apple macOS, among others. This chapter discusses the process, iterations and changes the project has undergone. From the idea conception through the design process to the technology stack being used.

2.2.1 Architecture and Technologies

As presented in figure 4 of the current chapter, the application user interface is built with VueJS [VueJS n.d.(b)], a progressive web framework with the leverage of ElectronJS [OpenJS Foundation n.d.] which makes it possible to share one code base, build and distribute for different platforms.

ElectronJS embeds Chromium and NodeJS into its binary. While Chromium is an open-source browser project in charge of displaying the content built with VueJS library, NodeJS is the server runtime environment that adds extensive APIs to access the file system and other OS features like networking and I/O. These characteristics are later used in different parts of the application, for instance, when accessing the file system to upload an application or when interacting with the platform API.

2.2.1.1 Application Architecture

The architecture of the front-end application is component-based [Dhaduk 2021, May 18]. As opposed to monoliths, it adopts a modular approach that decomposes front-end UI, taking advantage of benefits like flexibility, responsibilities decoupling, similar to micro-services at the back-end.

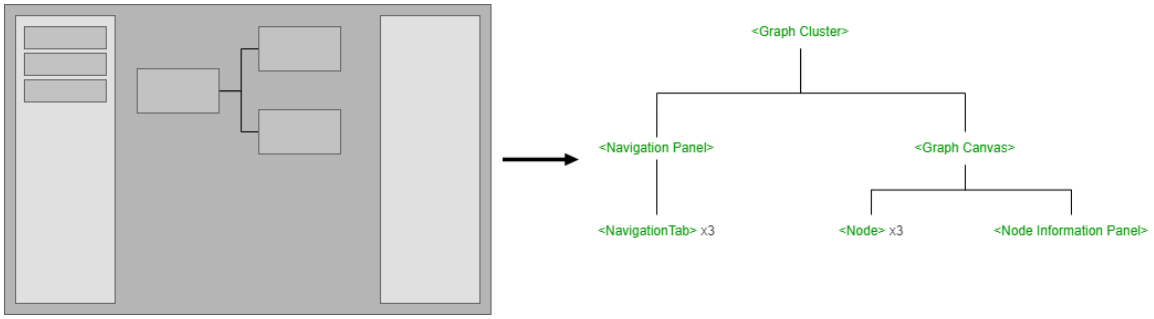


Figure 18: Component-based UI

Figure 18 illustrates the Cluster Graph page, one of the views from the YakoUI application. Using VueJS components, enables splitting the UI into independent and reusable pieces. The application is then organized into a tree of nested components.

2.2.1.2 State handling

To handle the UI data changes, VueJS, listens to data change events and re-renders the UI to display that change. The communication between components is through the propagation of properties. which works fine for smaller scale applications. However, due to the natural growth of the software, Vuex [VueJS n.d.(b)], a state management library has been used. The framework is the View component of the Flux pattern [Facebook n.d.].

To modify the values of this global centralized store, an event must be triggered from the view, which would be sent to the dispatcher. The dispatcher then broadcasts the resulting payload which updates the store.

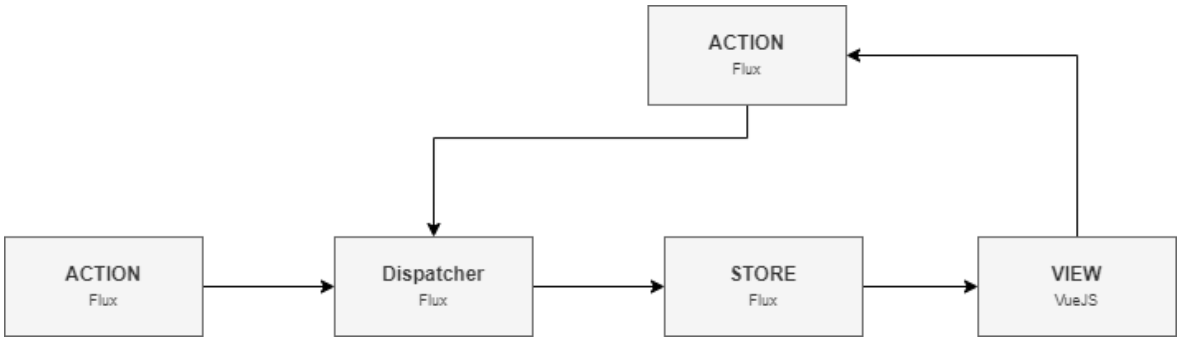
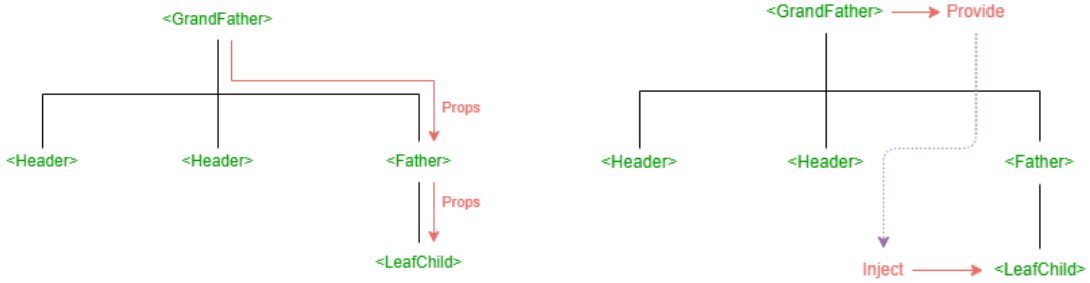


Figure 19: Vuex flux pattern

At some parts of the project, the Provide / Inject [VueJS n.d.(a)] feature has been used to pass data from the great-grandfather, namely any root component, to the leaf children components as shown in figure 20b, as opposed to components properties propagation which would lead to a prop-drilling issue.



(a) Prop-drilling effect

(b) Provide - Inject

Figure 20: Components properties propagation

2.2.2 Project Structure

The project source code structure is divided into 5 big folders:

- **Pages:** Contains the views of the application. For the final software, Upload Application page, Dashboard page and page were developed
- **Components:** Contains all the source code for the developed components for the project
- **Router:** Contains the logic for switching the pages of the application
- **Services:** Contains API endpoints to interact with the Yako platform and other utilities logic used across the software
- **Store:** Logic of Vuex, the global state management

The following UML diagram shows the relationship of the different components used in the software.

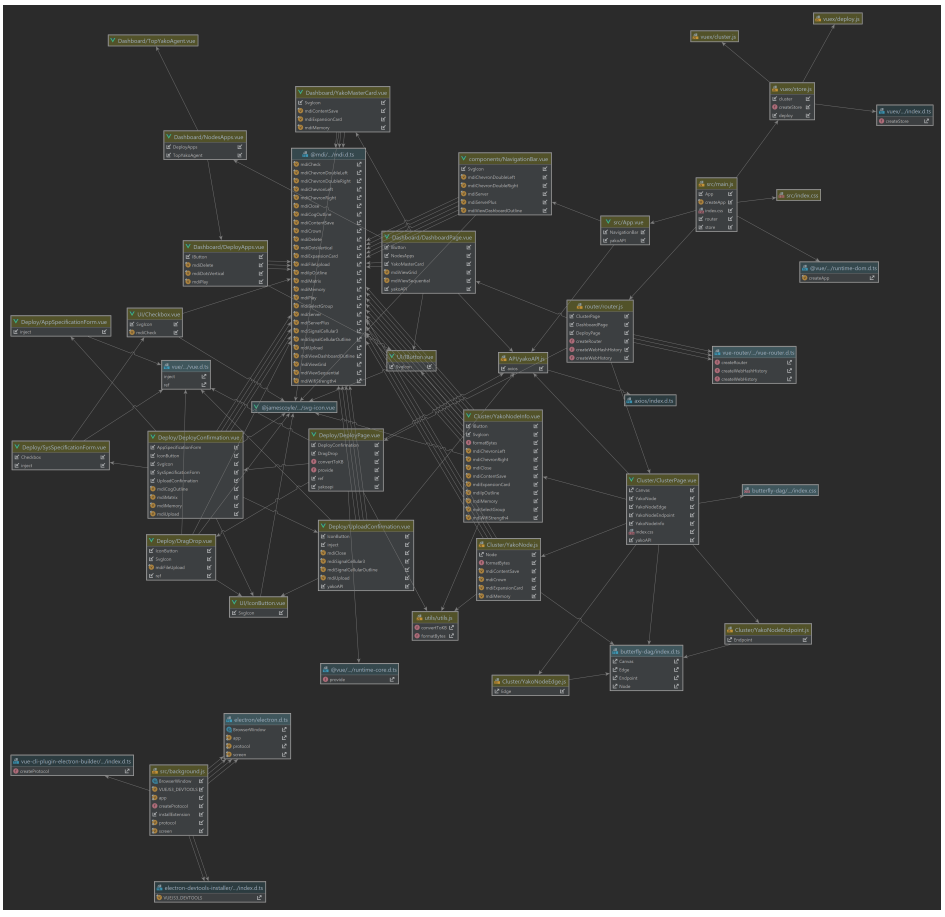


Figure 21: YakoUI project UML

2.2.3 Project Development

Project development sub-section covers the design phase of the application and the stages needed to bring this to life. The first iteration was to design wireframes. After this first iteration, more accurate UI designs were made. Which are the schemes of the look and feel of the final application. The last iteration involved converting the designs to code.

2.2.3.1 Stage 1: Wireframing

At the early stages of the project, mid-fidelity wireframes were designed to accurately represent the layout and clearly portray the information architecture of what would become the panel of the platform. In the following designs, user flow and functionalities are outlined.

2.2.3.1.1 Cluster graph wireframe

The cluster graph wireframe consisted in designing a view tab where the system administrator could view all the nodes (YakoMasters and YakoAgents) in the system, and how these are interconnected. The right side panel space is allocated to show additional information about the selected device.

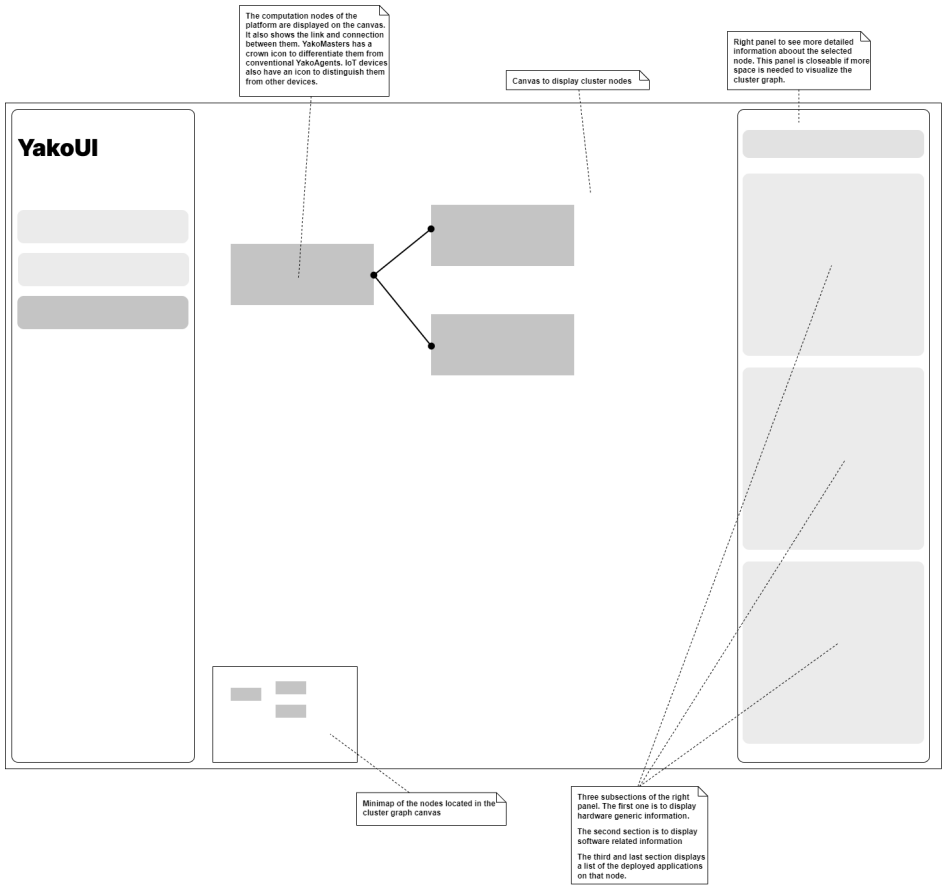


Figure 22: YakoUI cluster graph wireframe

2.2.3.1.2 Dashboard wireframe

This view tab is where all the information is gathered. The same nodes are shown in the left side of the page. On the top right side, a list of the top YakoAgent where the previously uploaded application could be uploaded are shown in descent order. The down right corner is reserved to display a list of the previously deployed application.

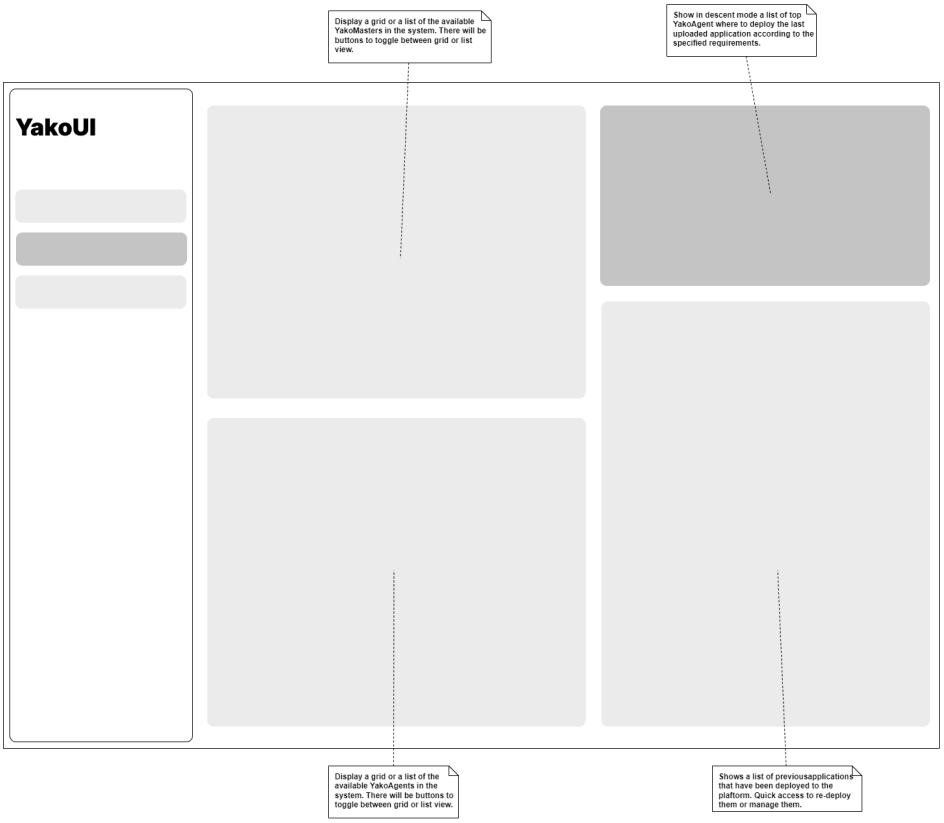


Figure 23: YakoUI dashboard wireframe

2.2.3.1.3 Upload and Deploy application wireframe

This view tab is the entry point of the platform. It will be shown on application start up, a drag and drop area is shown in the center where the system administrator could select the application to be deployed in the system.

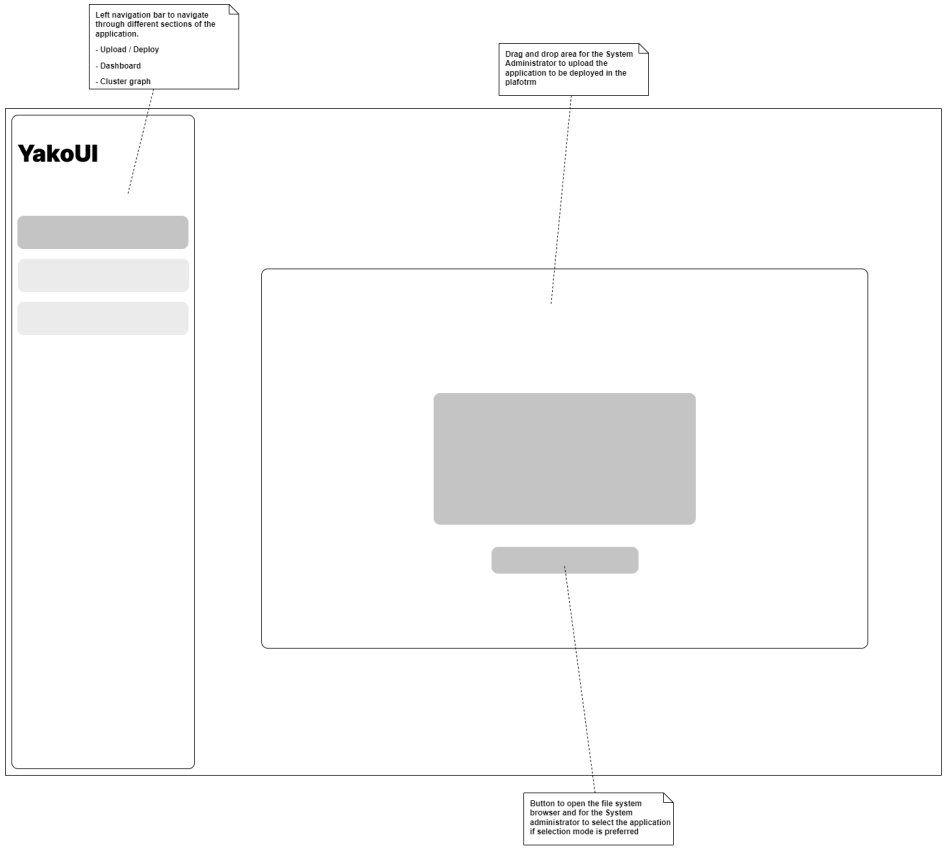


Figure 24: YakoUI deploy app wireframe

2.2.3.2 Stage 2: UI prototyping

In this second stage, all the previous low-fidelity wireframes were converted into UI prototypes. These includes more details on the sections. In this step a design system was defined.

2.2.3.2.1 Dashboard User Interface

The dashboard page provides a visual display of the platform. The view is divided horizontally in two sections, on the left side there are two subsections where the information about all the agents including the YakoMasters are displayed. For usability purpose, located on the top right corner of these sections two views styles buttons, grid and list, have been provided. Toggling between them will change the displaying format.

On the other half of this page, two other sections are reserved to display Top Nodes and the list of uploaded Applications. The node cards should display the agent identifier and the resources that comply with the requested system resources are highlighted in green, while those that do not fulfil are marked in red.

The enumerations of the Applications section are presented as a list of application cards. These cards show the application name on the left side, and buttons to interact with these are

situated on the other side.

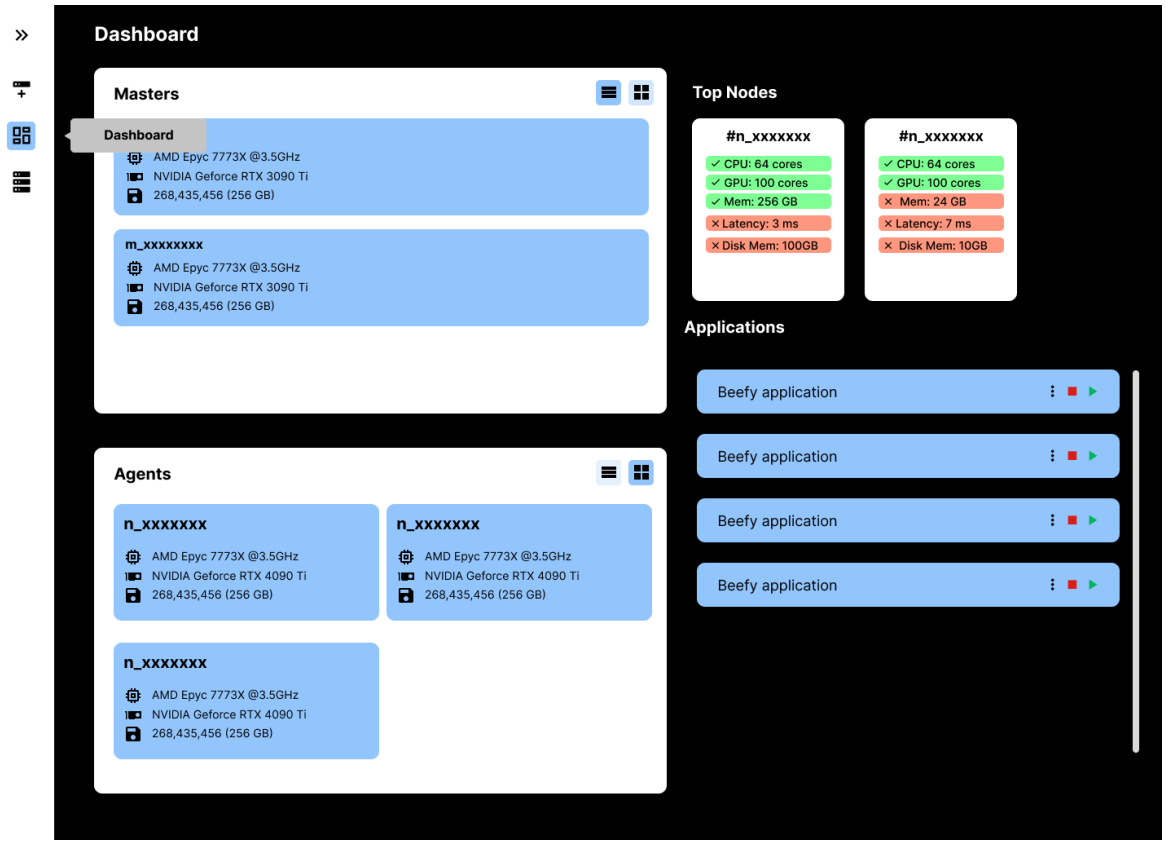


Figure 25: YakoUI dashboard UI

2.2.3.2.2 Cluster graph User Interface

The Cluster Graph page shows available nodes in a graphical mode. An extended version of the data is situated in the right side. It can be displayed at user's convenience by clicking on any of the nodes. This panel is divided into 4 main sections, General Information, System Information, Network Information and Deployed Applications lists shows the active running applications on that node.

An even more detailed version of the previously expanded right panel, figure 27, can be shown by clicking the left chevron tab situated in the left center part. On click it will enlarge the floating window showing the same amount of previous information with the addition of several graphs and telemetry in graphical mode.

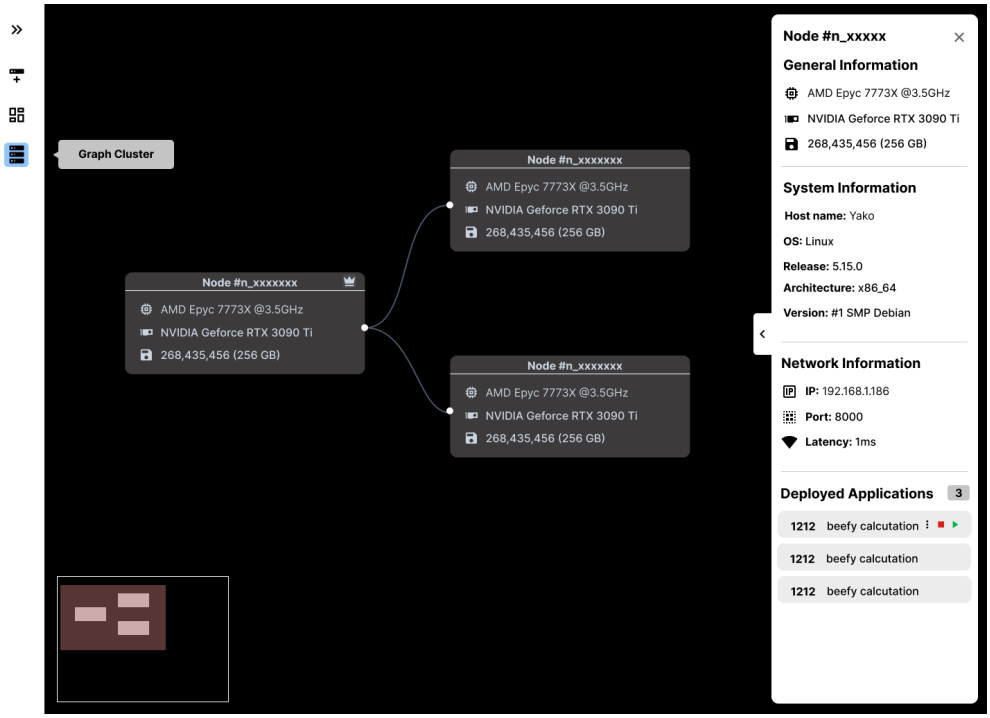


Figure 26: YakoUI Cluster Graph UI

This view, figure 27, was not developed for the delivered v1 software, due to time constraints and the prioritization of other major features.

Figure 27: YakoUI Expanded YakoAgent Information Panel UI

2.2.3.2.3 Upload and Deploy application User Interface

This is the first page that will be presented to the user on application start up. This view consists of a region where the system administrator can drag and drop the application to be deployed in the cluster. If manual selection is preferred, a "Select file" button is made available.

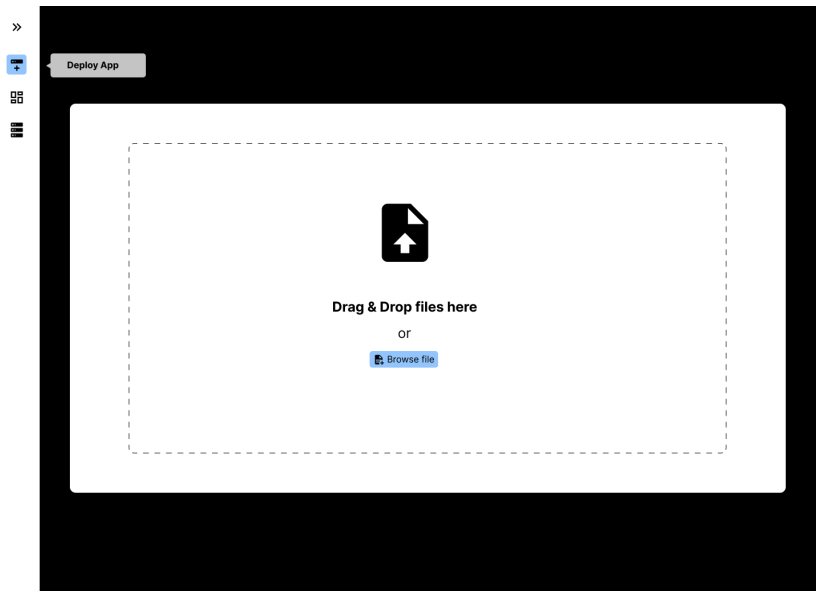
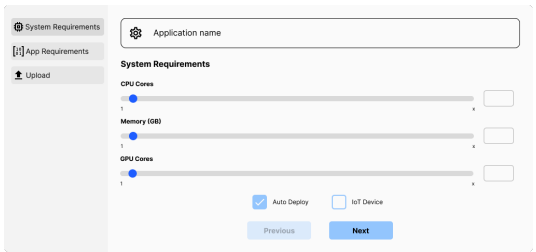
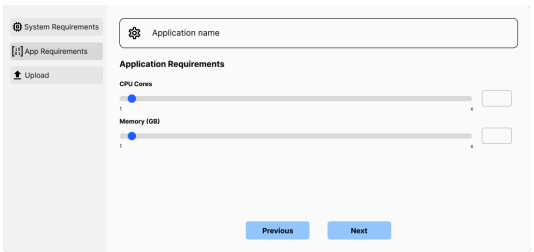


Figure 28: YakoUI Deploy Application UI

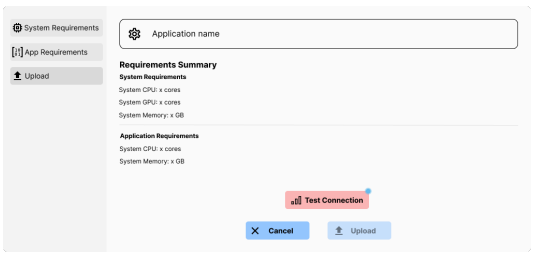
Once the application has been selected, a pop-up form will appear, figure 29.



(a) System Requirements form



(b) Application Requirements form



(c) Summary form tab

Figure 29: Upload application form

This window contains 3 tabs. The first asks the user to select all the preferred specifications of the system where the binary will be ran. These system requirements specifications will be collected to find the most suitable node of the cluster. The following tab collects resources for the virtualization of the application (The back-end logic for this feature could not been developed for release v1). The last tab summarizes all the selected properties. A connection test must be fired before being able to upload and deploy the application.

2.3 Run the system

This section describes the steps to run the Yako platform, both back-end and front-end software.

2.3.1 Run the back-end

2.3.1.1 Dependencies

Before being able to compile YakoMaster, YakoAgent and YakoAgent (IoT), projects tools and dependencies must be installed in the system. All three applications are programmed in Golang, Google's open-sourced programming language.

The following enumeration lists all the tools required to compile the platform.

1. Golang v1.17.x or greater
2. Make
3. Apache Zookeeper
4. gRPC
5. Protocol buffers compiler (Protobuf)
6. Eclipse Mosquitto MQTT Broker

2.3.1.2 Compilation

A Makefile is available in the source code repository. Table 5 lists all the predefined make rules.

Table 5: Makefile rules for make

Rule	Script	Description
gen_proto	protoc –proto_path=src/grpc/proto/ –go-grpc_opt = require_unimplemented_servers = false –go_out=src/grpc/ –go-grpc_out=src/grpc/ src/grpc/proto/.proto	Generates the protobufs code used by the gRPC client & server from the .proto source code
clean	rm src/grpc/pb/*.go	Cleans the generated protobuf files
build_master	go build src/yako_master/YakoMaster.go	Compiles YakoMaster
run_master	./src/yako_master/YakoMaster $(ip) $(port) $(zk_ip) $(zk_port) $(mqtt_ip) $(mqtt_port)	Runs the YakoMaster
build_agent	go build src/yako_node/YakoAgent.go	Compiles YakoAgent
run_agent	./src/yako_node/YakoAgent $(ip) $(port) $(zk_ip) $(zk_port)	Runs the YakoAgent
build_agent_iot	go build src/yako_agent_iot/YakoAgentIoT.go	Compiles YakoAgent (IoT)
run_agent_iot	./src/yako_agent_iot/YakoAgentIoT $(ip) $(port) $(mqtt_ip) $(mqtt_port)	Runs the YakoMaster (IoT)

2.3.1.2.1 gRPC and Protocol Buffers

gRPC stubs and procedures must be generated before proceeding with the project setup. Ensure that the system have these Go plugins installed, used for protocol buffers' compilation. This includes Golang gRPC and Golang Protocol Buffers compiler. Generate the Go gRPC source code by executing running the following code snippet.

```
go install google.golang.org/protobuf/cmd/protoc-gen-go@latest
go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@latest
export PATH="$PATH:$(go env GOPATH)/bin"
```

Source Code 16: Golang gRPC dependencies Installation

Yako platform software dependencies are located in the modules file go.mod, these must be installed first. Download these using snippet 17. Finally, to build all three programs run code snippet 18.

```
go mod tidy
```

Source Code 17: Download and install Golang Dependencies

```
# Builds YakoMaster
make build_master

# Builds YakoMaster
make build_agent

# Builds YakoAgent (IoT)
make build_agent_iot
```

Source Code 18: Yako platform compilation

2.3.1.3 Execution

Before running the Yako platform software, ZK and the Mosquitto MQTT broker must be fired up first.

2.3.1.3.1 Zookeeper

To start or stop ZK, one must access the directory where Apache Zookeeper has been installed. Otherwise, if the binaries are loaded into the system PATH running code snippet 19 will do the work.

```
# Run Zookeeper in current directory
./zkServer start

# Run Zookeeper from PATH binary
zkServer start

# Stop Zookeeper in current directory
./zkServer stop

# Stop Zookeeper from PATH binary
zkServer stop
```

Source Code 19: Start/Stop Apache Zookeeper

The ZK configuration, zoo.cfg file, is stored in the /conf directory relative to the installed ZK path. To monitor and troubleshoot ZK, the provided Java client or C binary can be used. An example of the configuration file 20 can be found in the project at Yako/src/utils/zookeeper/zoo.cnf. The highlighted line of code 20 should be replaced for the local ZK installation

logs folder or a preferred custom path. In this configuration ZK also fires an administrator panel up at port 8081, it can be accessed with a browser on <zk_ip>:8081/commands.

```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial synchronization phase can take
initLimit=10
# The number of ticks that can pass between sending a request and getting an
↪  acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
dataDir= /apache-zookeeper-x.x.x/logs
# the port at which the clients will connect
clientPort=2181
# Admin Server
admin.serverPort=8081
# the maximum number of client connections.
# increase this if you need to handle more clients
#maxClientCnxns=60
#
# The number of snapshots to retain in dataDir
#autopurge.snapRetainCount=3
# Purge task interval in hours
# Set to "0" to disable auto purge feature autopurge.purgeInterval=1

## Metrics Providers
#
# https://prometheus.io Metrics Exporter
#metricsProvider.className=
↪  org.apache.zookeeper.metrics.prometheus.PrometheusMetricsProvider
#metricsProvider.httpHost=0.0.0.0
#metricsProvider.httpPort=7000
#metricsProvider.exportJvmInfo=true
```

Source Code 20: Apache Zookeeper Configuration file

2.3.1.3.2 Mosquitto MQTT Broker

A configuration file is provided in the source code. The Mosquitto configuration file will set the broker to listen to the specified port, in the example below it starts the service at port 8002. For testing purposes, the broker also allows unauthenticated clients, Disable this property is recommended in production environment for security reasons. More specific documentation of Mosquitto broker can be found at its **official website**.

```
# Yako mosquitto broker configuration file
listener 8002
# Disable unauthenticated clients in production env
allow_anonymous true
```

Source Code 21: Mosquitto MQTT Configuration file

Run the following command to start the MQTT broker. Appending a & sign at the end makes the command run in the background.

```
mosquitto -c ./mosquitto.conf
```

Source Code 22: Start Mosquitto MQTT broker

2.3.1.3.3 YakoMaster

YakoMaster accepts a total of six arguments. The first two arguments, firstly the IP and secondly the port, are used to specify the socket of the machine where it will be run. The next two arguments are used to specify the socket of the ZK service registry machine that could be located in a separate physical machine. The final two arguments are needed to specify the location of the Mosquitto MQTT Broker.

- Make rule:

```
make run_master ip=<IP> port=<Port> zk_ip=<ZK IP> zk_port=<ZK Port> mqtt_ip=<MQTT
↪  IP> mqtt_port=<MQTT Port>
```

Source Code 23: YakoMaster make rule

- Manual deploy:

```
./YakoMaster <IP> <Port> <ZK IP> <ZK Port> <MQTT IP> <MQTT Port>
```

Source Code 24: YakoMaster manual deploy

2.3.1.3.4 YakoAgent

To deploy a YakoAgent, four arguments must be passed, the first two are the agent's machine IP and Port, while the last two are the location of the service registry. On connection, YakoMaster will be automatically notified.

- Make rule:

```
make run_agent ip=<IP> port=<Port> zk_ip=<ZK IP> zk_port=<ZK Port>
```

Source Code 25: YakoAgent make rule

- Manual deploy:

```
./YakoAgent <IP> <Port> <ZK IP> <ZK Port>
```

Source Code 26: YakoAgent manual deploy

2.3.1.3.5 YakoAgent (IoT)

To deploy a YakoAgent (IoT), four arguments must be assigned, the device's location and the Mosquitto MQTT broker socket.

- Make rule:

```
make run_agent_iot ip=<IP> port=<Port> mqtt_ip=<MQTT IP> mqtt_port=<MQTT Port>
```

Source Code 27: YakoAgent (IoT) make rule

- Manual deploy:

```
./YakoAgentIoT <IP> <Port> <MQTT IP> <MQTT Port>
```

Source Code 28: YakoAgent (IoT) manual deploy

2.3.2 Run the front-end

2.3.2.1 Dependencies

Firstly, ensure that the following two tools are installed in the system. The installation of NPM is generally bundled into NodeJS.

1. NodeJS
2. Node Package Manager (NPM)

2.3.2.2 Compilation

The front-end application's dependencies and other meta-data are located into the package.json. Firstly, installing these libraries is required. Secondly, table 6 lists all pre-defined commands to build the application.

```
# Install project dependencies
npm install

# Run NPM scripts
# E.g: To start  dev app server: npm run electron:serve
npm run <rule>
```

Source Code 29: Install project dependencies

Table 6: NPM scripts in package.json

Rule	Script	Description
serve	vue-cli-service serve	Starts a development server served on the browser
build	vue-cli-service build	Builds application to be served on the browser
lint	vue-cli-service lint	Analyse code for potential errors
electron:serve	vue-cli-service electron:serve	Starts a development application server
build-win	vue-cli-service electron:build –win –ia32 –x64 –publish=onTagOrDraft	Builds the desktop application for Windows OS 32-bits and 64-bits architecture
build-mac	vue-cli-service electron:build –mac –publish=onTagOrDraft"	Builds the desktop application for macOS
build-linux	vue-cli-service electron:build –linux –publish=onTagOrDraft	Builds the desktop application for Linux

For Windows OS systems, the application can be compiled as an installer or a portable executable for both 32-bits and 64-bits architecture systems. For MacOS, the bundled application is both instalable and portable. For Linux based system, different installers for different Linux distributions are generated. Please refer to the specific OS flavour official manual to install its packages.

3 Results

The initial plan was to carry the testing out in a testing bench provisioned by the university. Notwithstanding, due to external technical constraints, it end up not being possible. I finally decided to conduct all the tests from my machines. Tests were performed in my local cluster. The following tables lists the physical devices and virtualized machines used. The virtualization software used is VMWare Workstation 16 Pro.

3.1 Testing environment and setup

List of physical devices used in the test:

- Desktop Personal Computer
- Laptop
- Raspberry Pi 4 (4 GB)
- Raspberry Pi Zero W

The following sub-sections enumerate the components of the Yako platform, mentioned in section 2.1, and the location of these services.

3.1.1 Apache Zookeeper

ZK was run on my physical laptop with a x86_64 Intel i7-8550U CPU, operating under a Linux Kali rolling distribution. The service was available at the default port established in the configuration file 20.

Table 7: Apache Zookeeper service discovery service location

Device	IP	Port	OS
Laptop	192.168.1.41	2181	Kali Linux Rolling 2022.1

3.1.2 Mosquitto MQTT Broker

The MQTT broker was also run under the same physical device as ZK. The service is exposed at the port defined in snippet 21.

Table 8: Mosquitto MQTT Broker service location

Device	IP	Port	OS
Laptop	192.168.1.41	8002	Kali Linux Rolling 2022.1

3.1.3 YakoMaster

The YakoMaster orchestrator service was available on the same laptop. All, this and previously mentioned, services are running in the same physical device because of the limited testing devices available. Nevertheless, in a production environment, these could be deployed and ran in different physical machines. To all intents and purposes, it is advised to do so, to avoid SPOF [VMWare, Inc. n.d.] and increase security.

Table 9: YakoMaster and YakoAPI service location

Device	IP	Port	OS	CPU	RAM	GPU
Laptop	192.168.1.41	8001	Kali Rolling 2022.1	Intel i7-8550U	16GB	NVIDIA GTX 1050

3.1.4 YakoAgents

All the available YakoAgents were VMs, virtualized in both my desktop workstation and my portable computer.

Table 10: YakoAgents services locations

Device	IP	Port	OS	CPU	RAM
PC	192.168.1.44	8001	Ubuntu 20.04.4 LTS (Focal Fossa)	AMD Ryzen 7 3700X (2 Cores)	4GB
PC	192.168.1.45	8002	CentOS Stream 8	AMD Ryzen 7 3700X (1 Core)	1GB
PC	192.168.1.46	8003	Ubuntu 21.10 (Impish Indri)	AMD Ryzen 7 3700X (4 Cores)	8GB
PC	192.168.1.47	8004	CentOS Stream 9	AMD Ryzen 7 3700X (4 Cores)	12GB
Laptop	192.168.145.128	8005	Ubuntu 20.04.4 LTS (Focal Fossa)	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz (2 Cores)	4GB
Laptop	192.168.145.129	8006	CentOS Stream 8	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz (1 Core)	1GB
Laptop	192.168.145.130	8007	Ubuntu 21.10 (Impish Indri)	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz (4 Cores)	8GB

The heterogeneity components of these environments were elevated as much as possible, by setting different hardware configurations (CPU cores, RAM and Secondary Memory [HDD, SSD, NVMe SSD]), and differing software configurations. The OSes used in these machines are different versions of Linux Ubuntu (20.04 LTS and 21.10) and CentOS (Stream 8 and Stream 9).

3.1.5 YakoAgents (IoT)

On the other hand, YakoAgents (IoT) were physical Raspberry Pis devices. The Pi 4 was connected to the network through a gigabit switch and the Pi Zero W through a wireless connection (802.11 b/g/n) up to 600 Mbps. The more capable device booted from a 2.0 USB device while the other was ran from a micro SD (see Figure 30).

Table 11: YakoAgents (IoT) services locations

Device	IP	Port	OS	CPU	RAM
Raspberry Pi 4	192.168.1.35	8001	Raspbian 10 (Buster) 32-bits	ARMv7 processor rev 3 (v7l)	4GB
Raspberry Pi Zero W	192.168.1.76	8002	Raspbian 10 (Buster) 32-bits	ARMv6 processor rev 7 (v6l)	512MB

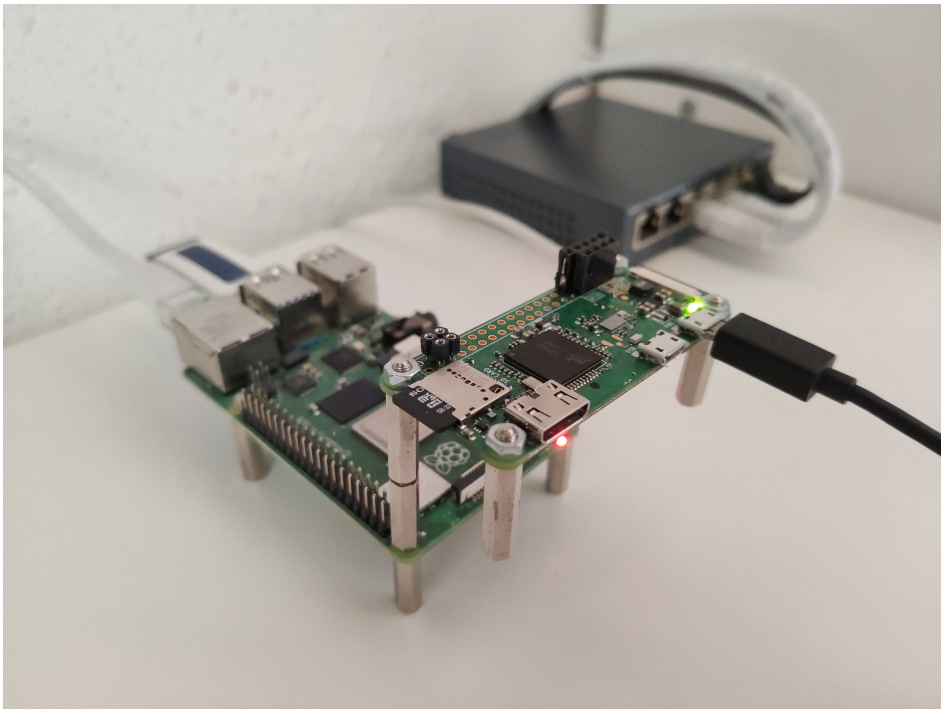


Figure 30: YakoAgents (IoT) Raspberry Pi physical devices

3.1.6 YakoUI

For this specific test shown in the figure 31, the front-end application was run on a Windows 11 device. However, previous testings were performed on Kali Linux. The web version is also accessible from a web browser at the hosting device served at port 8080 (see Figure

32). The service port can be configured on Webpack server configuration. In a production environment this should be on port 80 (HTTP) or 443 (HTTPS).

Table 12: YakoUI front-end application location

Device	IP	OS
PC	192.168.1.37	Windows 11 Insider Preview 25126.1000 (rs_prerelease)

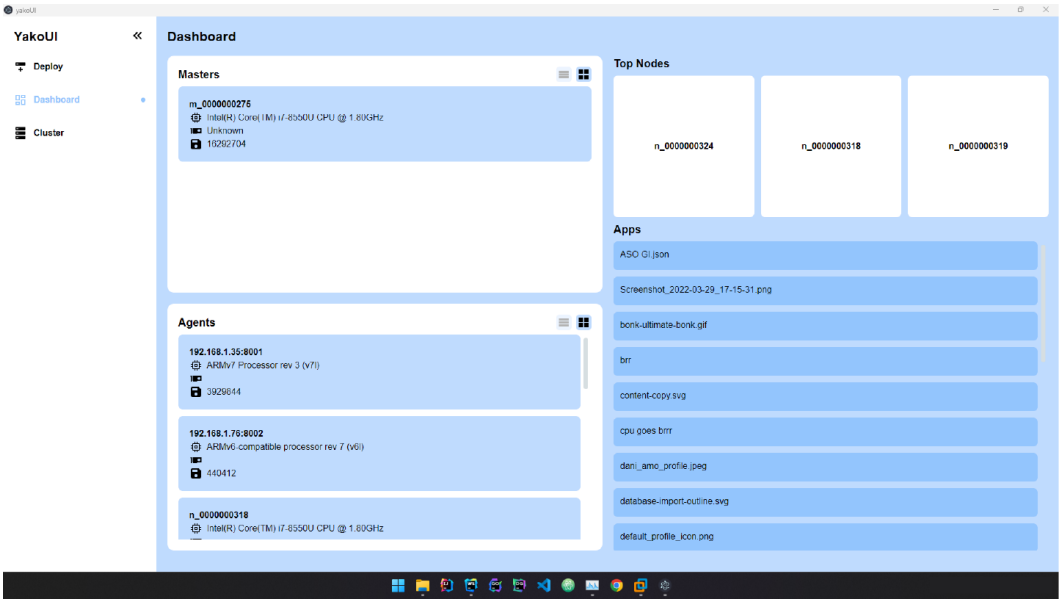


Figure 31: YakoUI application running on a Windows 11 device

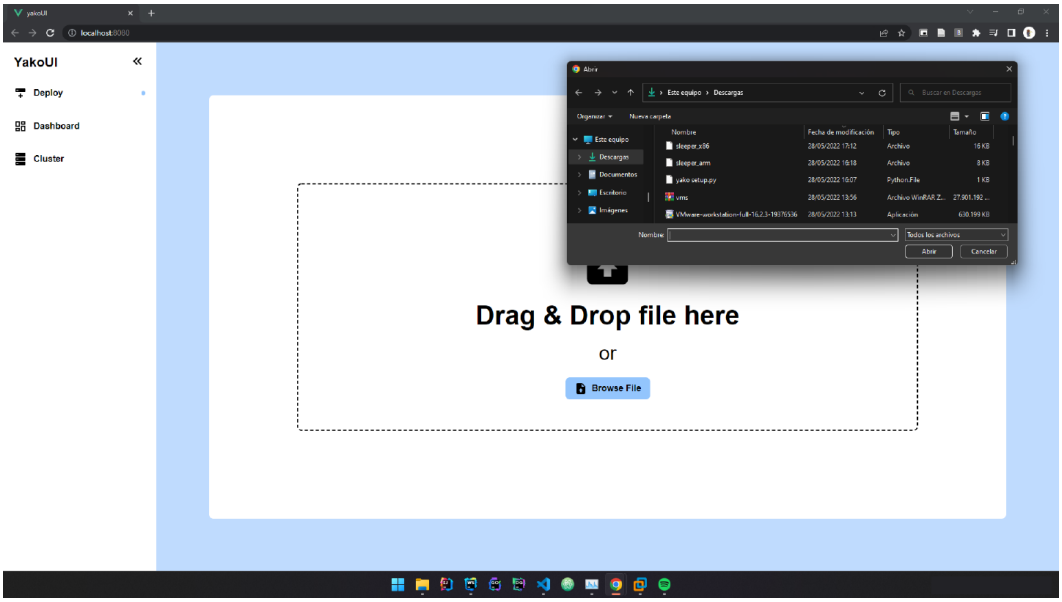
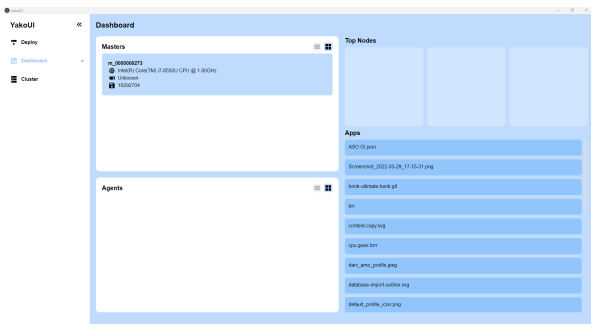


Figure 32: YakoUI application accessed from the web

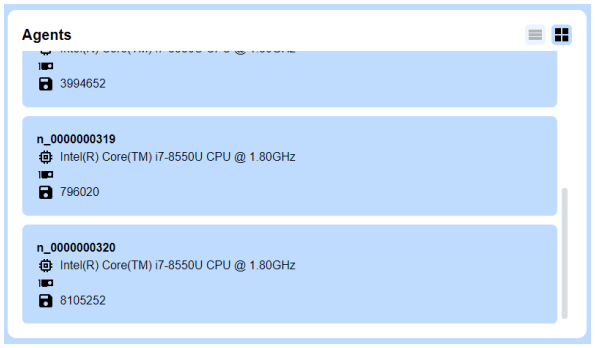
3.2 Testing procedure

On application startup the first view shown is the Upload application page. Before the upload of an executable to the system, agent devices must be connected first, figure 33a. During the testing stage, laptop VM YakoAgents were connected first, figure 33b. Physical Raspberries YakoAgents (IoT), figure 33c, were added following the prior devices connections. Lastly, PC VM YakoAgents were included to the platform as shown in 33d.

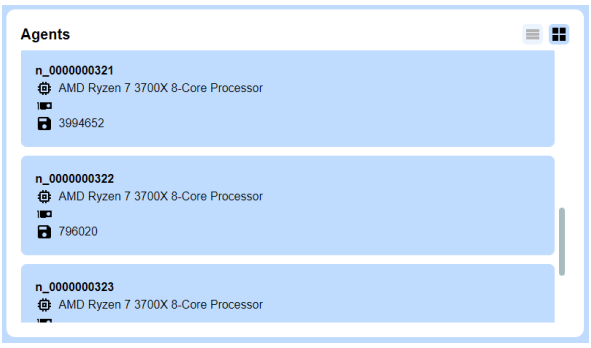


(a) No agents connected

(b) Laptop YakoAgent connections



(c) Pis YakoAgent (IoT) connections



(d) PC YakoAgent connections

Figure 33: YakoAgent and YakoAgent (IoT) devices connections

Figure 34 illustrates the resulting diagram of all connected devices in the platform from the Cluster graph view of the front-end application. The root node is the YakoMaster orchestrator, the first column after that are the IoT devices. The middle column are the 3 laptop VMs agents, and the last column are the 4 PC VMs.

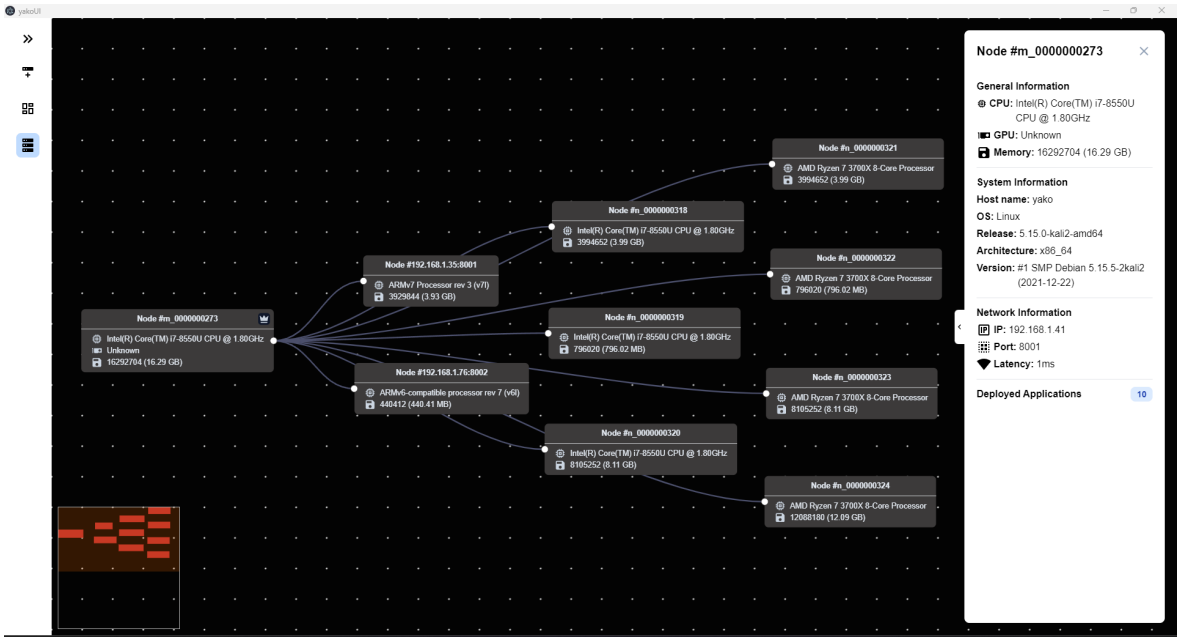


Figure 34: Test bed cluster graph

3.2.1 YakoAgent (IoT) Application deployment

The testing program, sleeper.c (see source code 30) was compiled into a binary executable called sleeper_arm. This small executable prints on a standard output sleeps for 5 seconds and finishes its process. The resulting compiled executable is platform and architecture dependent and should be taken into consideration. An application that has been compiled for a traditional desktop grade CPU architecture, like x86_64 would not work in an IoT ARM based device. The bits of the OS should also be taken into consideration. Currently, Yako platform does not support architecture and platform search, but it could be added in the future as a filter in the upload form.

```
#include <stdio.h>
#include <unistd.h>

int main() {
        while (1) {
                printf("Sleeping for 5 seconds\n");
                sleep(5);
                return 0;
        }
}
```

Source Code 30: Sleeper Once

Running the "file" command on the compiled versions of sleeper.c for x86_64 (see figure

31) and ARM (see figure 32) results in differing properties, despite the fact that both binaries are originated from the same source code.

```
file sleeper_x86

sleeper_x86: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically
↪  linked, interpreter /lib64/ld-linux-x86-64.so.2,
↪  BuildID[sha1]=b0dd44316207d972fd4473e04202cd5d67b325de, for GNU/Linux 3.2.0,
↪  not stripped
```

Source Code 31: 64-bits Sleeper binary for x_86_64 "file" command output

```
file sleeper_arm

sleeper_arm: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically
↪  linked, interpreter /lib/ld-linux-armhf.so.3, for GNU/Linux 3.2.0,
↪  BuildID[sha1]=db3215dd328d08d3ccb77e7cdde6cf30014e34b5, not stripped
```

Source Code 32: 32-bits Sleeper binary for ARM "file" command output

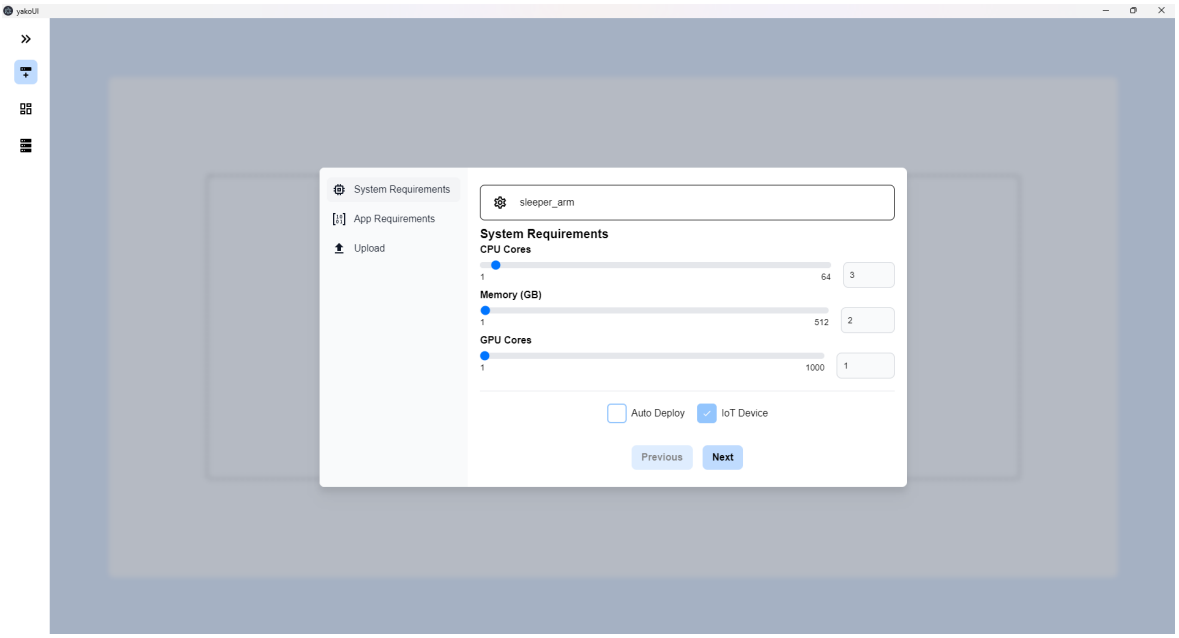


Figure 35: System requirements form to deploy a 32-bits app to ARM based IoT devices

Back to the Upload Application page, the executable was loaded in the front-end application. Following, the system requirements form was presented and the selection was an IoT node that should have 3 CPU cores and 2 GB of available RAM.

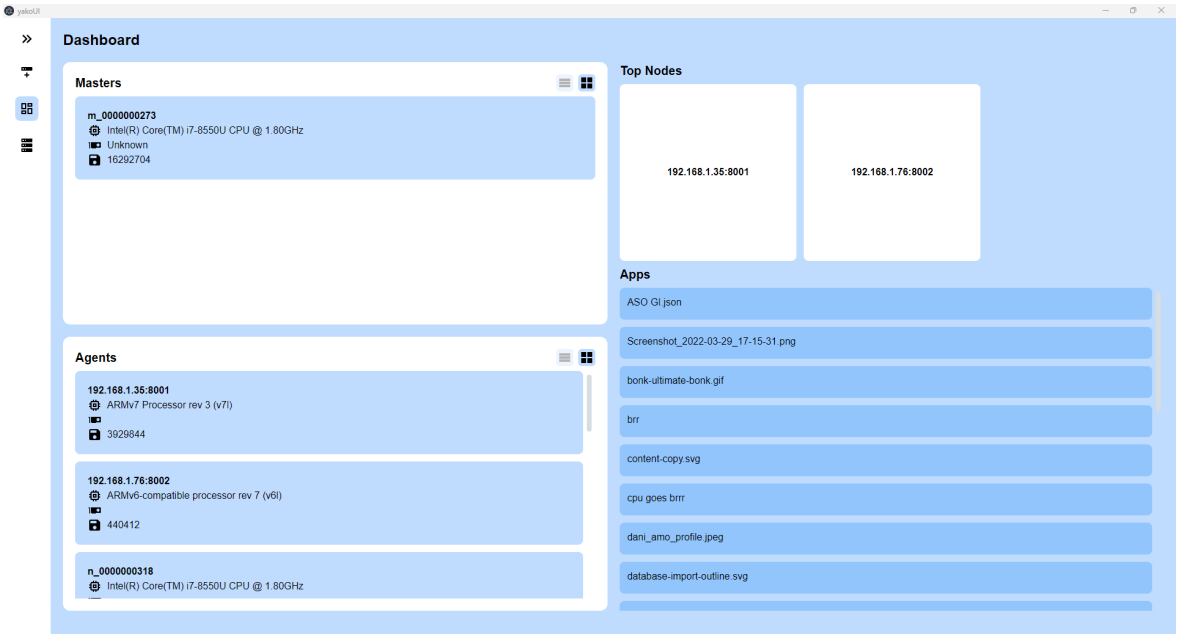


Figure 36: Top IoT ARM based nodes for a 32-bits app complying the specified system requirements

After the upload of the application and requisites form, YakoMaster the orchestrator searches for the most suitable agent and presents these back to the front-end application Dashboard page. For this particular test, the application was deployed to the node residing at 192.168.145.35, the Raspberry Pi 4 which has 4GB of RAM and 4 CPU cores, since the other IoT device had lower specifications.

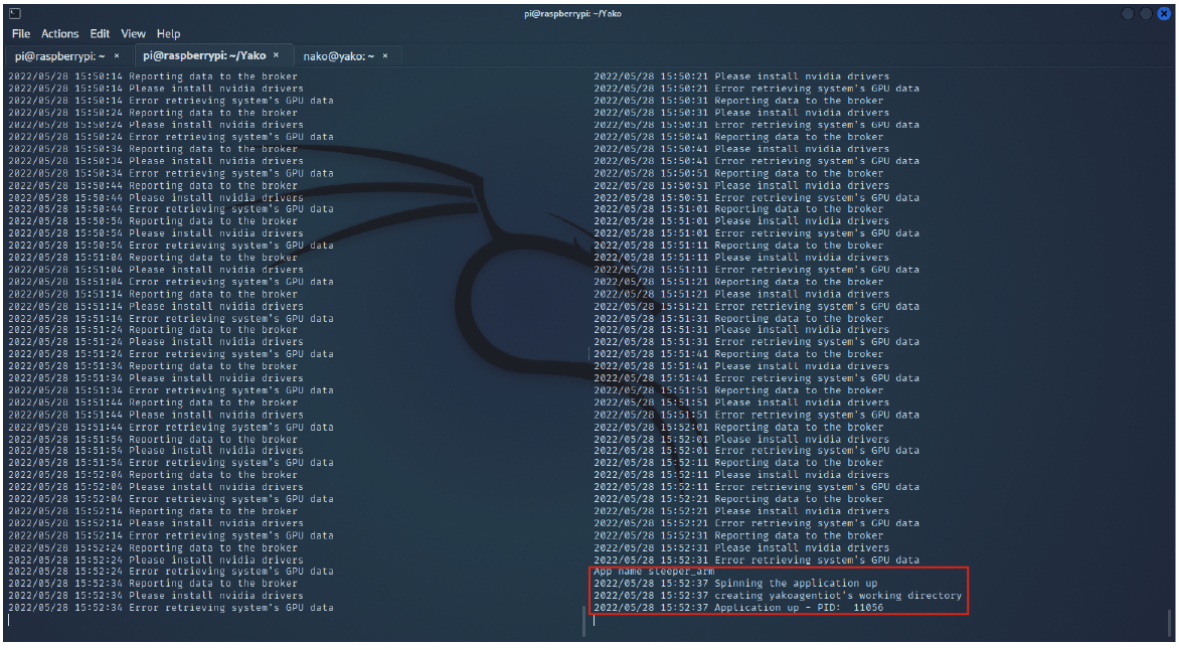


Figure 37: Deployed application to IoT device log

The application was deployed correctly to the corresponding agent. The process was started on the node with PID: 11056.

Figure 38: Deployed application to IoT device process

3.2.2 YakoAgent Application deployment

A slightly changed version of the previous sleeper program (see snippet 33) was compiled as sleeper_x86. Unlike the other, this version of the program will keep working until a kill signal is received. This application was dropped into the import box from the Upload Application page.

```c
#include <stdio.h>
#include <unistd.h>

int main() {
        while (1) {
                printf("Sleeping for 5 seconds\n");
                sleep(5);
        }
        return 0;
}
```

Source Code 33: Sleeper

In the system requirements form a node that was selected. The IoT Device checkbox was left unchecked. a summary of the requirements are illustrated in figure 40. The binary is then uploaded to the platform.

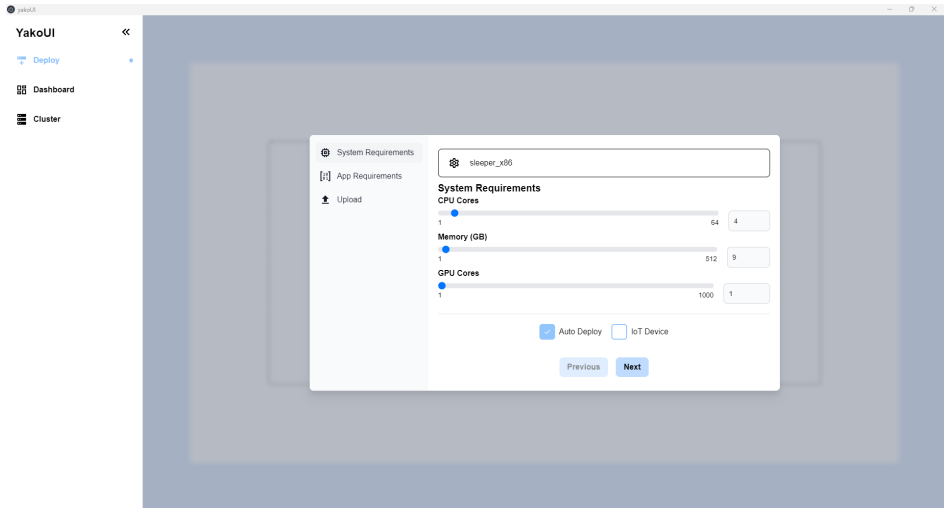


Figure 39: System requirements form to deploy a 64-bits app to x86_64 based devices

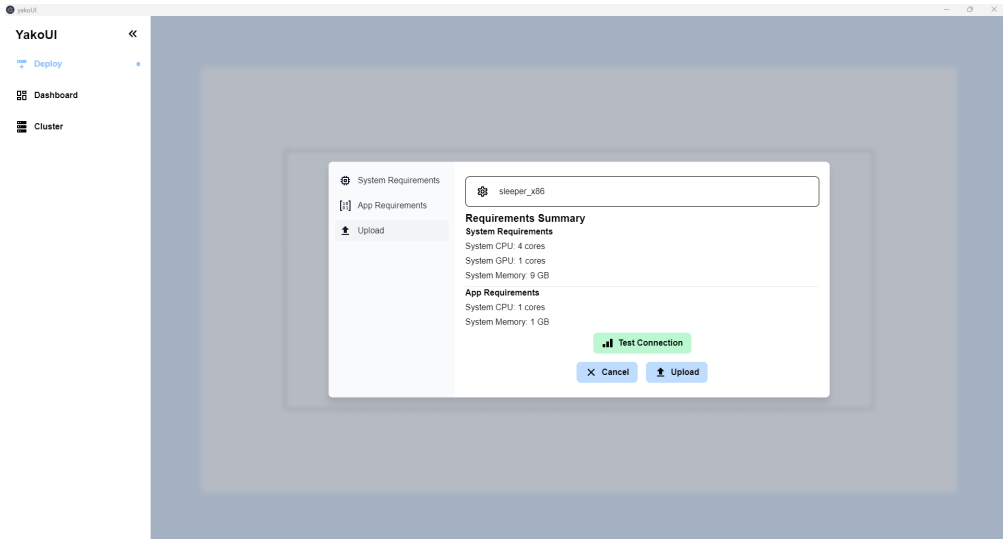


Figure 40: Summarized deployment requirements

On deployment request, the orchestrator will proceed to check the uploaded system requirements form and perform a search of the most suitable agent to deploy the application. In this test, IoT devices were discarded, and the agent selection was concluded among the remaining YakoAgents. The system requirements for this test was for the node to have 4 CPU cores and a total of 9GB of available free RAM. The application was finally deployed to YakoAgent n_0000000324 which corresponds to the PC VM Yako4. As shown in the debugging and profiling picture 42, this node has a total of 4 CPU cores and 12GB of RAM allocated, of which $\sim$10GB are available. It obtained a score of 2 brownie points, which positions the node as the most suitable one the run the app.

Figure 41: Top 3 YakoAgents for the 64-bits app

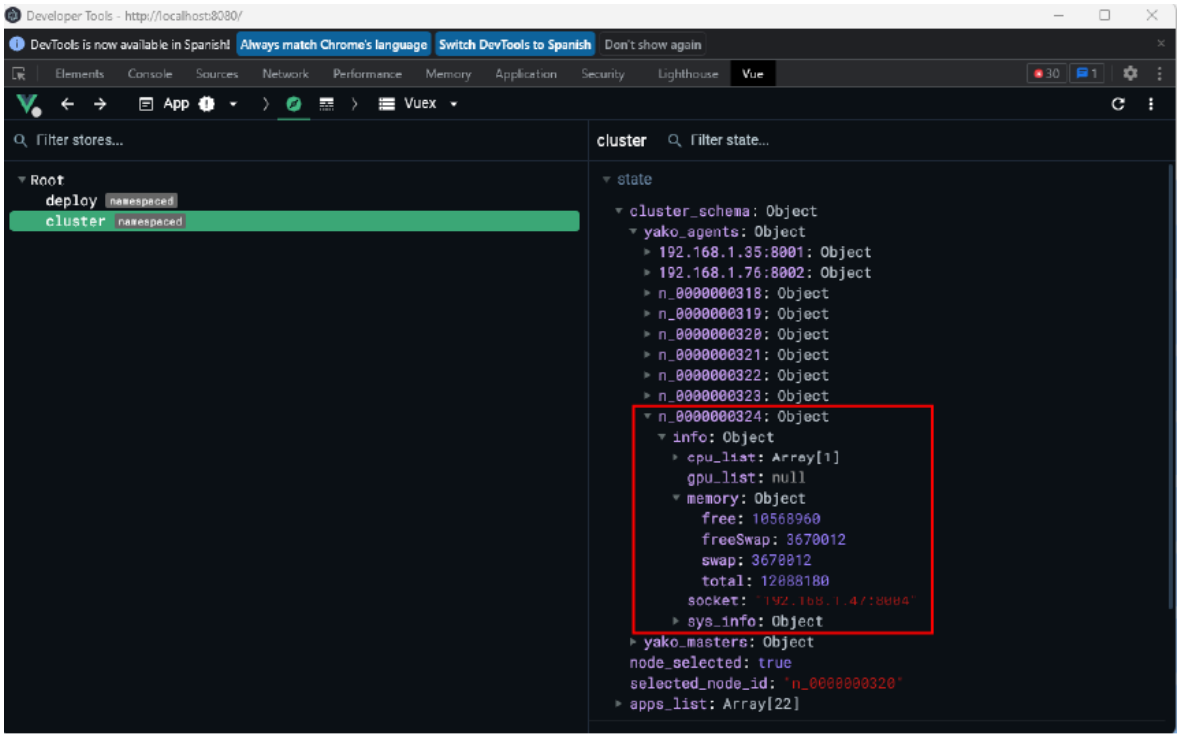


Figure 42: Top node system resources debug

As illustrated in figure 43, the application was transmitted correctly via gRPC streaming. The software was also successfully started with PID: 3707. On the right terminal the process was shown after running a quick process search.

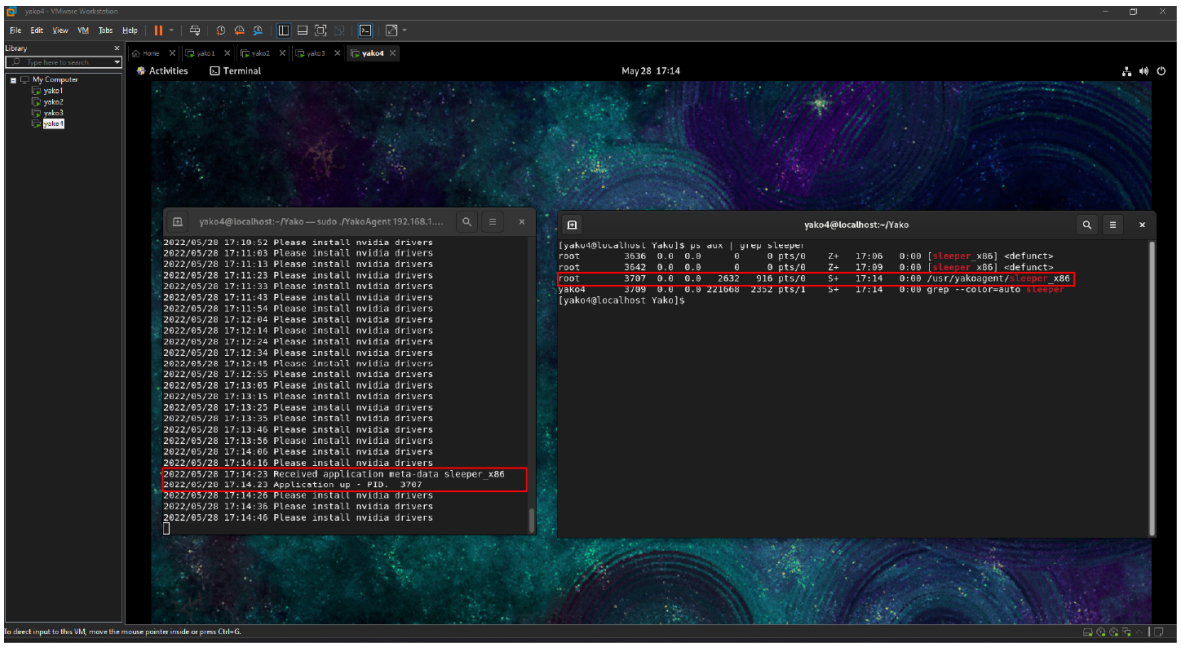


Figure 43: Deployed application to YakoAgent

4 Discussion

In light of the above, the Yako orchestrator works sublimely with the existing simple heuristic and deploys applications across the nodes of the cluster. Before examining the final set of nodes, the orchestrator ignores those that do not match the requirements. As shown in the previous section examples, a deployment of an application to IoT devices will discard the other non-IoT. The current algorithm examines the viability to distribute an application to a node at deploy time, depending on the current state of its resources. However, more advanced and complete heuristics, as described in the previous paragraphs, or AI based algorithms could be taken into consideration to further boost its smartness. For instance, using an AI-based technique could use stored information of the previous runs and predict the feasibility of the new deployment on that node.

Regarding the UI, currently the front-end application pages, (i) Dashboard (ii) Cluster Graph, only display basic telemetry information in a text-based manner. Adding the already designed expanded right panel view (see figure 27), could further improve the insight of the cluster state, for instance, its working agents and the resources being utilized. As stated in [Stark 2020, June 10], data visualization can provide users, in this case the system administrator, a clearer idea of what the information means by using maps or graphs. New components are under development, such as uploaded applications management and front-end application settings. Such changes are publicly available on GitHub repository [Chen 2022c, May 9].

During the internal testings, the Yako platform performed correctly, all its nodes, (i) YakoMaster, (ii) YakoAgent, (iii) YakoAgent (IoT), worked with full availability with no disconnections. Nevertheless, for the application to be production-ready more testing coverage should be undergone. This reinforces the need of considering using a bigger scale heterogeneous distributed system and more advanced testing techniques [Neeru360 2021, February 22], including QA, performance monitoring, stress loading and use case tests.

5 Conclusions and future work

5.1 Conclusions

In this project, a functional smart device-agnostic orchestrator has been developed to address the increasing adoption of heterogeneous hardware and software in the cluster environment.

The main focus of the project was on the elaboration of a MVP software that could be expanded in the future. Its modular architecture and decoupling of concerns permits new features to be added as new requirements appear.

In summary, the work provides system administrators an easy-to-use front-end to interact with an orchestrator with support for service discovery and matching for application deployment.

From my personal point of view, I culminated all the knowledge and abilities acquired during the undergraduate into this final project. The achievement of this has been achieved thanks to the knowledge learned during this period. In following table 13, the relation of multiple subjects' concepts applied to the different sections of the project is shown.

Table 13: CS bachelor subjects and its applications in the Yako platform

CS Undergraduate subject	Usage in the project
Design and Usability	Wireframing and YakoUI front-end application UI design
Data Structures and Algorithms	Back-end orchestrator software algorithms
Web programming I and II	Front-end application and API interfacing
Local Area Networks and Networks Interconnection	Services interconnection, sockets and networking interfacing
Distributed Systems	Yako platform architecture
System Design and Administration	Metrics extraction, Linux administration
Software Methodologies I and II	Sequence diagrams, UML and software patterns
Operating Systems and Advanced Operating Systems	Sytem resources and processes threading
Project Management and Agile	Project planning and scheduling

5.2 Future work

For this project two pieces software have been developed, during its realisation many features was presented on the design phase. However, some of those ended not being included to the MVP for being neither critical nor imperative. The following paragraphs cover the future line of work for the Yako platform.

Features to be improved or added to the back-end software:

- Support for YakoMaster nodes fault tolerance using leader election algorithm [Amazon, Inc. n.d.(b)] with the already implemented Apache Zookeeper. The Yako platform would be able to operative even if a YakoMaster stops working.
- One feature that was placed to one side, was the support of virtualized deployments with the Linux kernel cgroups [Foundation n.d.]. Virtualizing software not only adds a

new layer of security but also isolation at the hardware level. In that case YakoMaster would also become a high-level hypervisor that would be able to control the allocation of resources.

- Port the YakoAgent to other OS platforms to support more diversity.
- Add support for more filters and system requirements for a smarter deployment node selection.
- Add more endpoints to manage resources like application re-deployment post-upload or to query specific agent's information.
- Sandboxing [Check Point n.d.] the applications to test against malware before the real deployment.
- Including an authentication system would prevent unauthorized users to interact with the platform.

Different aspects of the front-end software could also be improved:

- As mentioned in the previous section, better UI and more visual widgets could be developed to represent the insights the orchestrator provides. This makes the data more natural for the human mind to comprehend and therefore makes it identify patterns.
- Redirecting agent's logging to the front-end and reporting errors notification. Currently these are shown in their respective nodes consoles.

6 Acknowledgement

I would like to express my gratitude to both my supervisors, Dr. Joan Navarro Martín and Dra. Anna Carreras Coch, who mentored and offered guidance throughout the entire project. Dr. Daniel Amo Filvà deserves my thanks for proofreading the paper. I would also like to thank Ms. Watanabe Rika for supplying help to my inquiries and Mercari, Inc. for providing me with a premium Udemy account for my personal learning and growth, which I used during the realisation of the dissertation.

References

Academic. (n.d.). *List of CPU architectures* [Academic dictionaries and encyclopedias]. Retrieved May 21, 2022, from https://en-academic.com/dic.nsf/enwiki/11834151

Agile Manifesto. (n.d.). *Manifesto for agile software development*. Retrieved May 30, 2022, from https://agilemanifesto.org/

Amazon, Inc. (n.d.[a]). *Fully managed container solution – amazon elastic container service (amazon ECS) - amazon web services* [Amazon web services, inc.]. Retrieved May 31, 2022, from https://aws.amazon.com/ecs/

Amazon, Inc. (n.d.[b]). *Leader election in distributed systems* [Amazon web services, inc.]. Retrieved May 30, 2022, from https://aws.amazon.com/builders-library/leader-election-in-distributed-systems/

Camp, T. (n.d.). *Guide to pub/sub in golang | ably blog: Data in motion* [The ably blog]. Retrieved May 22, 2022, from https://ably.com/blog/pubsub-golang

Check Point. (n.d.). *What is sandboxing? - check point software*. Retrieved May 30, 2022, from https://www.checkpoint.com/cyber-hub/threat-prevention/what-is-sandboxing/

Chen, J. (2022a, May 4). *Yako* [original-date: 2021-12-27T15:55:42Z]. Retrieved May 31, 2022, from https://github.com/JiahuiChen99/Yako

Chen, J. (2022b, May 4). *Yako project* [original-date: 2021-12-27T15:55:42Z]. Retrieved May 31, 2022, from https://github.com/users/JiahuiChen99/projects/1

Chen, J. (2022c, May 9). *YakoUI* [original-date: 2022-01-29T18:04:19Z]. Retrieved May 31, 2022, from https://github.com/JiahuiChen99/YakoUI

Cloud Native Computing Foundation. (n.d.). *Operating etcd clusters for kubernetes* [Kubernetes] [Section: docs]. Retrieved May 17, 2022, from https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/

Conventional Commits. (n.d.). *Conventional commits* [Conventional commits]. Retrieved May 31, 2022, from https://www.conventionalcommits.org/en/v1.0.0/

cURL. (n.d.). *Curl*. Retrieved May 19, 2022, from https://curl.se/

Dhaduk, H. (2021, May 18). *Component-driven development: Best practices to build scalable frontend.* [Insights on latest technologies - simform blog]. Retrieved May 17, 2022, from https://www.simform.com/blog/component-based-development/

Docker. (2022, May 31). *Swarm mode overview* [Docker documentation]. Retrieved May 31, 2022, from https://docs.docker.com/engine/swarm/

Eclipse. (2018, January 8). *Eclipse mosquitto* [Eclipse mosquitto]. Retrieved May 22, 2022, from https://mosquitto.org/

Facebook. (n.d.). *Flux | flux*. Retrieved May 26, 2022, from https://facebook.github.io/flux/

Fernandez-Alonso, E., Castells-Rufas, D., Joven, J., & Carrabina, J. (2012). Survey of NoC and programming models proposals for MPSoC. *International Journal of Computer Science Issues*, *9*.

Foundation, T. L. (n.d.). *Control group v2 — the linux kernel documentation*. Retrieved May 30, 2022, from https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html

Gin Gonic. (n.d.). *Gin web framework* [Gin web framework]. Retrieved May 31, 2022, from https://gin-gonic.com/

Guides, S. (n.d.). *Scrum guide | scrum guides*. Retrieved May 30, 2022, from https://scrumguides.org/scrum-guide.html

Hoppscotch. (2022, May 20). *Hoppscotch/hoppscotch* [original-date: 2019-08-21T13:15:24Z]. Hoppscotch. Retrieved May 20, 2022, from https://github.com/hoppscotch/hoppscotch

Karl-Bridge-Microsoft. (n.d.[a]). *Performance counters - win32 apps*. Retrieved May 27, 2022, from https://docs.microsoft.com/en-us/windows/win32/perfctrs/performance-counters-portal

Karl-Bridge-Microsoft. (n.d.[b]). *Process status API - win32 apps*. Retrieved May 27, 2022, from https://docs.microsoft.com/en-us/windows/win32/psapi/process-status-helper

MacKenzie, D. (n.d.). *Uname(1): Print system info - linux man page*. Retrieved May 31, 2022, from https://linux.die.net/man/1/uname

MDN. (n.d.). *Content-type - HTTP | MDN*. Retrieved May 21, 2022, from https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Type

Meena, J. S., Sze, S. M., Chand, U., & Tseng, T.-Y. (2014). Overview of emerging non-volatile memory technologies. *Nanoscale Research Letters*, *9*(1), 526. https://doi.org/10.1186/1556-276X-9-526

Microsoft Corporation. (n.d.). *What is cloud computing? a beginner's guide | microsoft azure*. Retrieved May 30, 2022, from https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/

Mistretta, P. J., & Gault, R. (n.d.). Micron's automata processor, 28.

Mittal, S., & Vetter, J. S. (2015). A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys*, *47*(4), 1–35. https://doi.org/10.1145/2788396

Neeru360. (2021, February 22). *Software testing techniques* [GeeksforGeeks] [Section: Software Engineering]. Retrieved May 30, 2022, from https://www.geeksforgeeks.org/software-testing-techniques/

Netflix Technology Blog. (2017, April 18). *Netflix shares cloud load balancing and failover tool: Eureka!* [Medium]. Retrieved May 17, 2022, from https://netflixtechblog.com/netflix-shares-cloud-load-balancing-and-failover-tool-eureka-c10647ef95e5

OASIS. (n.d.). MQTT version 5.0. Retrieved May 7, 2022, from https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

OpenAPI Initiative. (n.d.). *OpenAPI* [OpenAPI initiative]. Retrieved May 31, 2022, from https://www.openapis.org/

OpenJS Foundation. (n.d.). *Electron | build cross-platform desktop apps with JavaScript, HTML, and CSS.* Retrieved May 31, 2022, from https://www.electronjs.org/

Postman. (n.d.). *Postman API platform* [Postman]. Retrieved May 19, 2022, from https://www.postman.com/

Rajaee, O. (2017). IoT, resource constrained devices, security. https://www.researchgate.net/publication/316490787_IoT_Resource_Constrained_Devices_Security#:~:text=The%20majority%20devices%20that%20will,handle%20extra%20functionalities%20and%20protocols.

RedHat. (2020, May 8). *¿Qué es una API de REST?* Retrieved May 31, 2022, from https://www.redhat.com/es/topics/api/what-is-a-rest-api

Reed, B., & Kalmár, N. (2019, September 16). *PoweredBy - apache ZooKeeper - apache software foundation*. Retrieved May 9, 2022, from https://cwiki.apache.org/confluence/display/ZOOKEEPER/PoweredBy

Stark, M. (2020, June 10). *Why data visualization is important* [Analytiks]. Retrieved May 30, 2022, from https://analytiks.co/importance-of-data-visualization/

VMWare, Inc. (n.d.). *What is a single point of failure? definition & FAQs* [Avi networks]. Retrieved May 31, 2022, from https://www-stage.avinetworks.com/glossary/single-point-of-failure/

VueJS. (n.d.[a]). *Provide / inject | vue.js*. Retrieved May 31, 2022, from https://vuejs.org/guide/components/provide-inject.html

VueJS. (n.d.[b]). *What is vuex? | vuex*. Retrieved May 26, 2022, from https://vuex.vuejs.org/

Zahran, M. (2017). Heterogeneous computing: Here to stay. *Communications of the ACM*, *60*(3), 42–45. https://doi.org/10.1145/3024918