LUCAS URZEDO DA SILVA PAULINO

Advisor: Joubert de Castro Lima

ANY.JS: A RESTFUL GENERAL-PURPOSE COMPUTING MIDDLEWARE FOR CLOUD ENVIRONMENTS

Ouro Preto
October of 2022

Federal University of Ouro Preto
Institute of Exact Sciences and Biology
Undergraduate Program in Computer Science

Monograph presented to the Undergraduate Program in Computer Science of the Federal University of Ouro Preto in partial fulfillment of the requirements for the degree of Bachelor in Computer Science.

LUCAS URZEDO DA SILVA PAULINO

Ouro Preto
October of 2022

FEDERAL UNIVERSITY OF OURO PRETO

Any.JS: A RESTful general-purpose computing middleware for cloud environments

LUCAS URZEDO DA SILVA PAULINO

Dr. Joubert de Castro Lima – Advisor
Federal University of Ouro Preto

Bachelor Gabriel de Oliveira Ribeiro – Co-Advisor
CI&T

Ouro Preto, October of 2022

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

Lucas Urzedo da Silva Paulino

Any.JS: A RESTful general-purpose computing middleware for cloud environments

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 27 de Outubro de 2022.

Membros da banca

Joubert de Castro Lima (Orientador) - Doutor - Universidade Federal de Ouro Preto
Gabriel de Oliveira Ribeiro (Coorientador) - Bacharel - CI&T
André Luiz Lins de Aquino (Examinador) - Doutor - Universidade Federal de Alagoas
André Luís Barroso Almeida (Examinador) - Mestre - Instituto Federal de Minas Gerais

Joubert de Castro Lima, Orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 27/10/2022.

Documento assinado eletronicamente por **Joubert de Castro Lima**, **PROFESSOR 3 GRAU**, em 28/10/2022, às 08:20, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0416027** e o código CRC **5F23293A**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35400-000
Telefone: 3135591692 - www.ufop.br

Abstract

Middleware is everywhere and most likely will remain everywhere for a long time because it helps reduce the complexity of distributed application development. Middlewares with RESTful APIs are very useful because they are interoperable and designed for Web. Service-oriented middleware architectures turn the solutions extensible, isolated, scalable and sometimes a bit easier to guarantee fault tolerance in containerized environments due to their modularity. Many RESTful service-oriented IoT middleware solutions were developed in recent years, but few general-purpose alternatives were presented by the literature and this is an important lacuna. A general-purpose middleware must run any algorithm, i.e., call any function or method transparently, as well as store variables and data structures also transparently over a cluster. The existing RESTful APIs for processing and storage services have several limitations: i) no support for non-primitive data types as function parameters; ii) no class/component registration service; iii) no observability service to notify the caller about the end of storage and processing asynchronous calls; iv) no data structure iterator service; and v) no lock/unlock service to guarantee data structures and variables concurrent and safe accesses. The consequence is a limitation of the REST services usage, requiring a language specific complement to fully explore processing and storage services. In this work, we present the Any.JS tool, a RESTful general-purpose computing middleware for cloud environments with the support for the previous five mentioned limitations of the literature. Thus, no complementary language specific API calls are necessary with Any.JS, since only interoperable JSON objects and HTTP requests/responses are sufficient to provide general-purpose services. A comparative analysis were made against Ignite and Hazelcast solutions to highlight the strengths and weakness of Any.JS and its counterparts. Any.JS was slower than the existing solutions, but it supports a larger catalog of RESTful services. As we can see, many improvements are mandatory for the existing literature to achieve both efficiency and interoperability.

Keywords: Middleware. General-purpose Computing. RESTful API. REST. Service-oriented Architecture. Cloud Computing. Ignite. Hazelcast.

Contents

1 Introduction **1**
1.1 Goals 2
1.2 Out of Scope 3
1.3 Work Organization 3

2 Basic Concepts **4**
2.1 Cloud Computing 4
2.2 Virtual Machine 4
2.3 Application Programming Interface 5
2.4 REST and RESTful API 5
2.5 Microservices 6
2.6 Service Orchestration and Choreography 6
2.7 Container 6
2.7.1 Docker 7
2.7.2 Docker Swarm 7
2.8 MongoDB - No SQL storage 7
2.9 KrakenD gateway 8

3 Related Work **9**
3.1 General-purpose solutions 9
3.2 Middleware solutions for other purposes 12
3.3 Discussions 12

4 Development **14**
4.1 Architecture 14
4.1.1 The Virtual Machine (VM) Layer 15
4.1.2 Container Orchestrator 15

4.1.3 Containers 15
4.1.4 Services 17
4.1.5 API Gateway 19
4.1.6 Any.JS REST API 19
4.1.7 Any.JS Clients 26
4.2 The Any.JS API usage example 27
4.3 Any.JS example used against Ignite 30
4.4 Any.JS example used against Hazelcast 33

5 Experiments **36**
5.1 VMs setup 36
5.2 Containers setup 36
5.3 Metrics 37
5.4 Any.JS versus Ignite 38
5.4.1 Any.JS Experimental Results 38
5.4.2 Ignite Experimental Results 39
5.4.3 Comparative Analysis 40
5.5 Any.JS versus Hazelcast 41

6 Conclusion **43**

Bibliography **45**

List of Figures

4.1 Any.JS architecture 14
4.2 Any.JS container types 16

5.1 Any.JS, Hazelcast and Ignite container configurations 37

List of Tables

3.1 Any.JS and its counterparts' features 12

5.1 VMs configuration 36
5.2 Any.JS Binary Objects runtimes with 1 to 4 VMs and containers 38
5.3 Any.JS Binary Objects CPU usage with 1 to 4 VMs and containers 39
5.4 Any.JS Objects runtimes with 1 to 4 VMs and containers 39
5.5 Any.JS Objects CPU usage with 1 to 4 VMs and containers 39
5.6 Ignite synchronous with 1 to 4 VMs and containers 40
5.7 Ignite asynchronous with 1 to 4 VMs and containers 40
5.8 Any.JS Java runtimes 41
5.9 Any.JS JavaScript runtimes 41
5.10 Hazelcast Java runtimes 41

Acronyms

API Application Programming Interface. iii, iv, 5, 19–26, 43

AuFS Advanced Multi-Layered Unification Filesystem. 7

BSON Binary JSON. 18

CLI Command Line Interface. 17

CPU Central Processing Unity. 36

CRUD Create, Retrieve, Update and Delete operations. 2, 20

GCP Google Cloud Platform. 36

HTTP Hypertext Transfer Protocol. 5, 20

IP Internet Protocol. 36

JSON JavaScript Object Notation. 5, 17, 20

LXC Linux Container. 7

OS Operating System. 4–6, 15, 36

RAM Random Access Memory. 36

REST Representational State Transfer. 5, 20, 26

SOAP Simple Object Access Protocol. 5

URL Uniform Resource Locator. 19, 20

vCPU Virtual CPU. 36

VM Virtual Machine. iii, vi, 4, 6, 15, 16, 36, 43

XML Extensible Markup Language. 5

Chapter 1

Introduction

Middleware is everywhere and most likely will remain everywhere for a long time because it helps reduce the complexity of distributed application development Al-Jaroodi and Mohamed (2012a). The challenging issue is to provide sufficient generalized high-level mechanisms for general-purpose services using middleware solutions and a rapid development of distributed and parallel applications. We have specific-purpose middleware solutions, like IoT Razzaque et al. (2015); Ngu et al. (2016), context-aware Kjær (2007); Vahdat-Nejad (2014), robotics Elkady and Sobh (2012) and Integration ones Marpaung et al. (2013); Jahantigh et al. (2020), and general-purpose ones Taboada et al. (2013); Almeida et al. (2019), where the last are designed for processing and storage services, precisely to run any algorithm or call any function/method with any number and various types of parameters. Besides that, the general-purpose solutions also instantiate and store variables and distributed data structures over a cluster transparently.

In recent years, service-oriented middleware solutions were presented Al-Jaroodi and Mohamed (2012b), being many of them designed for IoT requirements Issarny et al. (2016) and others with RESTful APIs Larian et al. (2022). The idea of an architectural design based on services and microservices enables modularity, extensability, manutenability, scalability and sometimes a better way to provide fault tolerance when services are deployed into containers and managed by orchestrators, like Docker Swarm Naik (2016) or Kubernetes Hightower et al. (2017). The RESTful API is a way to expose the services where the first benefit is the interoperability, since the service can be called by multiple applications developed into multiples programming languages using JSON objects. Besides that, the service is also designed for Web technology, thus HTTP protocol is adopted to provide requests and responses. In general. POST, GET, PATCH, DELETE and PUT HTTP

requests are sufficient to develop Create, Retrieve, Update and Delete operations (CRUD) microservices.

Unfortunately, few existing general-purpose middleware solutions implement RESTful APIs. We have found only Ignite Zheludkov et al. (2017), Hazelcast Veentjer (2013), Infinispan [1], Oracle Coherence Seovic et al. (2010) and RAFDA Walker et al. (2010), the last not really a RESTful solution, but also a Web service alternative. They implement storage and processing services, but they still have some limitations in some useful REST services, requiring specific programming languages complements for the following issues:

- No support for non-primitive or non-String data types as function parameters;
- No class/component registration service;
- No observability service to follow storage and processing asynchronous calls;
- No data structure iterator service; and
- No lock/unlock service to guarantee data structures and variables concurrent and safe accesses.

1.1 Goals

To design and implement a general-purpose middleware solution with a RESTful API with the following services: i) support for non-primitive data types as function parameters; ii) class/module registration service; iii) observability service to follow storage and processing asynchronous calls; iv) data structure iterator service; and v) lock/unlock service to guarantee data structures and variables concurrent and safe accesses.

Besides these core services, the middleware solution must provide a batch task execution service to perform many task runs over a cluster with a single API call. The results of a batch execution should be available also asynchronously, thus individually for each task run of a batch.

Finally, the presented alternative must be evaluated against one or more middlewares of the literature using JavaScript or other programming language to see the solutions performance, thus we can measure the impacts, but also see the benefits of new REST services for the general-purpose middleware literature.

[1] Infinispan - <https://infinispan.org/>

1.2 Out of Scope

It is not the scope of this work non general-purpose middleware solutions, like IoT, Cloud Integration, Context-awareness, Robotic, Wireless Sensor Network (WSN) and so forth. In the related work section, we mention some of them that were published in survey papers, but they were not evaluated against our solution, named Any.JS, nor analyzed accordingly.

1.3 Work Organization

The rest of this work is organized as follows: Chapter 2 discusses the basic concepts for a better understanding of the work. Chapter 3 presents the related work about middlewares. Chapter 4 details the architecture and design of the Any.JS middleware solution. Chapter 5 details the experiments and experimental results. Finally, in Chapter 6 the conclusion and future research directions are described.

Chapter 2

Basic Concepts

In this chapter, all the basic concepts that are useful for a better understanding of the work are presented.

2.1 Cloud Computing

Cloud Computing offers a shared set of flexible and configurable computing resources, such as network, CPU, GPU, RAM, hard disk, but also software layers, such as Operating System (OS), basic software (Ex. database, compilers, etc.) and applications (Ex. stream processing tools, graph tools and so forth). These resources can be easily deployed and used at scale with minimal efforts Mell et al. (2011).

A single hardware can be abstracted as several small devices or a cluster of machines can be abstracted as a single device. The elasticity is another fundamental property in cloud environments, so the user can increase the processing or storage capacities, as well as any other hardware or software configuration on-the-fly, thus cloud computing became very useful for high availability demands.

2.2 Virtual Machine

The Virtual Machine (VM) is the core of cloud computing and it is an emulated computer system with all the layers (hardware, OS and applications) running over another computer system. This architectural design provides efficiency and isolation for users Goldberg (1974).

A hypervisor (or virtual machine monitor - VMM) is a computer software, firmware or hardware that creates and runs VMs. A computer on which a hypervisor runs one or more VMs is called a host machine, and each VM is called a guest machine. The hypervisor presents the guest OS with a virtual operating platform and manages the execution of the guest OS. Multiple instances of a variety of OS may share the virtualized hardware resources. Any application can be deployed over such OS, being also virtualized.

2.3 Application Programming Interface

The Application Programming Interface (API) is used by computer systems for their communications, thus it defines how to make requests, the used data types, and how to interact to allow system communication [1]. The API can be customized for specific utilization, such as interoperability or transaction services.

In Web context, the API use Hypertext Transfer Protocol (HTTP) protocol to send the content of requests and responses. The common used format is the Extensible Markup Language (XML) or JavaScript Object Notation (JSON). The communication mechanism of Web APIs are migrating from Simple Object Access Protocol (SOAP) to Representational State Transfer (REST), thus a transaction or a request can take long periods to occur (hours, days or even months), a fundamental requirement for justice services, government services, scientific experiments and many more.

2.4 REST and RESTful API

Representational State Transfer (REST) is a set of constraints used by HTTP requests and responses to meet the guidelines defined in the software architecture. RESTful refers to an API adhering to those constraints. In a RESTful Web service, requests made to a resource's URI elicit responses with a payload formatted in HTML, XML, JSON, or some other format. RESTful systems aim for fast performance, reliability, and scalability by reusing components that can be managed and updated without affecting the system as a whole, even while it is running.

[1] <https://en.wikipedia.org/wiki/Application_programming_interface>

2.5 Microservices

One of several definitions of Microservices is:

"A Microservice is an independently deployable component of bounded scope that supports interoperability through message-based communication. Microservice Architecture is a style of engineering highly automated and where software systems are made up of capability-aligned microservices." Nadareishvili et al. (2016)

Microservices are small in size, being more cohesive, if compared with a monolithic architecture. Web APIs are often used to intercommunicate microservices. This software architecture enables the development of low coupling components and each of them can be developed at different stage. Normally, each microservice has its metadata that is very useful for both service discovery and service composition, a fundamental activity of service orchestration explained further in this chapter.

2.6 Service Orchestration and Choreography

Service orchestration refers to a centralized process responsible to interact and manage microservices. The interactions happen at message level and the orchestration includes business logic, including service composition, and task execution order Peltz (2003), this way the microservices must be flexible and adaptable to business needs.

Choreography services represent a decentralized architecture, where each participant, i.e., each microservice has a communication pattern focused in collaboration, thus elections and other decision strategies are used Busi et al. (2005). The same orchestration responsibilities (service discover, service composition, task ordering, etc.) are included in choreography, but in a distributed way.

2.7 Container

The container is also a virtualization layer and used together with VMs, since the containers operate at OS level Merkel (2014). In containers, there are abstraction layers for process, network, hard disks and so forth. Containers running on the same machine preserve the isolation, thus memory, processor and disk are not visible among them.

As we can see, containers are more light than VMs, this way faster during deployment, creation and destruction of computer environments. The VM requires kernel level, OS

libraries, drivers and many more, so even when it is not running it consumes a huge amount of memory. Normally, containers are fault tolerant, so if a containerized application crashes for any reason, the container can call any procedure to restart it. The recover service in terms of backlogs and operations ordering are domain application responsibilities.

2.7.1 Docker

Docker is one of the market-leaders container technology, which uses few features from Linux kernel to provide a lightweight tool to manage the life-cycle of many applications Merkel (2014). Docker uses Linux containers and Linux Container (LXC), a package from Linux containers to provide user-namespaces, separating container's database and network from host. Docker adopts Advanced Multi-Layered Unification Filesystem (AuFS) as its file system. It is a layered file system, enabling Docker containers to deploy several customized images derived from the same image. The LXC also provides *cgroups*, a package used to both analyze and limit containers resources, such as memory usage, disk space and I/O bandwidth.

Docker also provides a Web repository where there are images from several frameworks, OS, and programming languages (Ex. Python and MongoDB images). The most common way to manage Docker containers is through command line, but it also has a REST API.

2.7.2 Docker Swarm

Docker Swarm is a clustering tool, orchestrating Docker containers (nodes) into a virtual Docker system Naik (2016). It adopts nodes to provide a redundancy system in the case of processing failure. The Docker Swarm has Manager and Worker nodes, where the manager node allows the cluster state management, precisely the node deploy, update, and remove services in an existing cluster of nodes. On the other hand, the worker nodes are responsible for running tasks.

2.8 MongoDB - No SQL storage

MongoDB is a powerful, flexible, and scalable general-purpose database. It combines the ability to scale out with features, such as secondary indexes, range queries, sorting, aggregations, and geo-spatial indexes Chodorow (2013). The MongoDB, as Non-relational database, uses the document concept to storage the rows or registers. The document has

no predefined schemas, this way the document keys do not require prefixed types and sizes, thus they allow insertions or removals of fields more freely.

2.9 KrakenD gateway

KrakenD KrakenD (2021) is an extensible, declarative, high-performance open-source API gateway. Its main functionality is to create an API that acts as an aggregator of many microservices into single endpoints. It performs some operations automatically: aggregate, transform, filter, decode, throttle and authentication.

Chapter 3

Related Work

In this chapter, it is detailed the most similar middleware solutions designed to provide processing and storage services using RESTful APIs. These solutions are named general-purpose ones, thus specialized alternatives used for context-awareness, robotics, sensing, actuating or cloud integration services are mentioned, but they are not the focus of this work.

We have searched the following paper repositories to build the related work: IEEE and ACM. The Google search engine was also used to find white papers and magazine reports. The keywords searched were:

$$(middleware\ OR\ framework\ OR\ library)\ AND\ (rest\ OR\ restful)\ AND\ (api)$$

The number of papers or pages found were:

ACM library: 62 results (41 research papers, 7 posters, 5 short papers, 4 abstracts, 2 tutorials, 1 survey and 2 demonstrations);

IEEE Xplore: 133 results (125 conference papers, 5 journal papers, 2 magazine papers and 1 early access paper);

Google: 199.000 pages in English.

3.1 General-purpose solutions

Apache Ignite Zheludkov et al. (2017) is an open source distributed database for high-performance computing (HPC) with in-memory speed. It has a RESTful API for processing

and storage services. The processing API allows the user to develop custom tasks in contemporary languages, such as Java, C, C++, Python, JavaScript and PHP. The tasks run over the cluster and their results are available transparently, simplifying the development of parallel applications that demand high processing. Unfortunately, only String task parameters are allowed in the RESTful alternative. No register of classes in the cluster is provided via RESTful services, but just via ordinary programming languages alternatives and only inside the cluster. Ignite supports processing and storage faults, it has load balancing transparency and it can run tasks in a broadcast way or it can execute them on specific cluster nodes. The storage RESTful support does not implement lock and unlock to guarantee durability during a safe access. Remote instantiation is not feasible in Ignite, so only remote storage is possible using its RESTful API. Only distributed maps, named caches in Ignite, are possible, so common variables are not allowed. It can be deployed over public containerized cloud environments.

Hazelcast Veentjer (2013) is a streaming and memory-first application platform developed in Java. It is an open-source platform dedicated to distributed computing, which brings a set of components that support data distribution and processing. The solution offers support for different programming languages, such as C++, C, Java, .NET, Python, JavaScript and Go. For data distribution, Hazelcast has maps, multi-maps and collections. In terms of processing, the users can use locks and messages. The Executor framework is responsible for running tasks asynchronously, but this feature is only available in Java and not accessible on the RESTful API. Unfortunately, only String parameters are allowed in tasks submitted using RESTful alternative. Hazelcast handles map locks, ensuring that write operations on a given partition of the map are handled one at a time in first-in-first-out order, but via RESTful API the locks are not feasible. Data structures iterators can be implemented and used in Java, but not provided via RESTful services. Hazelcast's WAN Replication feature can be used via RESTful alternative to synchronize multiple clusters. Data can be imported from databases, files, messaging systems, on-premise and cloud systems in various formats using a RESTful service. Hazelcast offers pipelines and load/store interfaces for this purpose. Applications developed on clients can be asynchronous, but it was not mentioned about RESTful API calls for that purpose. It can be deployed over public containerized cloud environments.

Infinispan [1], developed by JBoss/RedHat Sliwinski et al. (2019), is a popular open source distributed in-memory <*key, value*> pair data store solution that enables accessing

[1] Infinispan - <https://infinispan.org/>

a cluster in two ways: i) via an API available in a Java library; ii) via several protocols, such as HotRod, REST, Memcached and WebSockets, making Infinispan a language-independent solution. In addition to storage services, the middleware can execute tasks remotely and asynchronously; however, developers must implement Runnable or Callable Java interfaces. Furthermore, it must register these tasks in the JVM classpath of each cluster node, which can delay the deployment process. Infinispan servers provide RESTful HTTP access to data through a RESTful endpoint. The RESTful API allows write and read data in different formats and Infinispan can convert between those formats when required. Through a RESTful client, it is possible to retrieve, execute, and upload Infinispan server tasks. Infinispan can be deployed into containerized cloud environments.

Oracle Coherence Seovic et al. (2010) is a Java-based distributed cache and in-memory data grid. Intended for systems that require high availability, high scalability and low latency, particularly in cases that traditional relational database management systems provide insufficient throughput, or insufficient performance. Coherence comes with its own Kubernetes operator, which allows to easily provision Coherence-based applications on any Kubernetes cluster. The Coherence RESTful API pre-defines many operations that can be used to interact with a cache. In addition, custom operations such aggregators and entry processors can be created as required. The RESTful services require metadata about the cache that they expose. Coherence RESTful uses HTTP as the underlying protocol and can marshal data in both JSON and XML representation formats. As we can see, its focus is on storage services, being not well designed for processing ones.

RAFDA Walker et al. (2010) is a reflective middleware that permits arbitrary objects in an application to be dynamically exposed for remote access, allowing applications write without concern with distribution. RAFDA objects are exposed as Web services to provide distributed access to ordinary Java classes. Applications access RAFDA functionalities using infrastructure objects called RAFDA Run-Time (RRT). Each RRT provides two interfaces to application programmers: one for local RRT access and the other for remote RRT access. With this approach, RAFDA introduces dependencies and, consequently, requires code refactoring. A RRT supports peer-to-peer communication; therefore, it is possible to execute a task in a specific cluster node. However, when developers need to submit several tasks to more than one remote RRT, they must implement a scheduler from scratch. The web service model provides no storage services, only processing ones, thus no data structures, locks/unlocks and iterators are available. The RRT can be used as a web services container and like conventional Web services containers, a list of available services

and the Web service description language (WSDL) for a particular service can be obtained from the RRT.

Table 3.1: Any.JS and its counterparts' features

Feature \ Tool	Fault Tolerance	Simple Deploy	Collaborative	Variable	Task	DS	DS traversal	SCA	Cloud Native
Any.JS	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Apache Ignite	Yes	No	Yes	No	Yes	Yes	Yes	No	Yes
Hazelcast	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes
Infinispan	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Oracle Coherence	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
RAFDA	No	Yes	Yes	Yes	Yes	No	No	No	Yes

3.2 Middleware solutions for other purposes

Besides the general-purpose middleware solutions, we have specific-purpose middleware solutions, like IoT Razzaque et al. (2015); Ngu et al. (2016), Cloud of Things (CoT) Farahzadi et al. (2018), context-aware Kjær (2007); Vahdat-Nejad (2014), service composition Ibrahim and Mouel (2009), robotics Elkady and Sobh (2012), authentication/authorization services Christie et al. (2020), wireless sensor network (WSN) Wang et al. (2008); Mohamed and Al-Jaroodi (2011), real-time computing Pérez and Gutiérrez (2014) and cloud integration Marpaung et al. (2013); Jahantigh et al. (2020).

In terms of integration, there are even semi-automatic ways of integrating legacy systems, IoT applications and cloud environments and an example is R2SMA Königsberger. and Mitschang. (2018). It is a middleware architecture to access legacy enterprise Web services using lightweight RESTful APIs. The solution implements a SOAP-to-REST middleware architecture that provides a semi-automatic way to create RESTful proxies from existing conventional Web services and without the need to adapt the current services.

3.3 Discussions

To the best of our knowledge, no general-purpose middleware solutions implement a minimum set of processing and storage services in their RESTful APIs. It is mandatory any sort of complement for the RESTful API calls using other APIs in specific programming languages to perform lock/unlock accesses, iterator data structure traversals, register of existing algorithms or components to be executed further and a way to observe future processing/storage runs to receive notifications when they finished. In terms of processing

services, only String or primitive data type parameters are supported via RESTful alternative. These limitations are the main drawbacks of the current general-purpose middleware literature. The support for several programming languages, their scalability, availability, their fault tolerance capacity and the deployment over elastic containerized cloud environments are the core strengths of the existing alternatives.

Another important consideration is the number of general-purpose players nowadays. We have found only five solutions and most of them are commercial ones, being almost of them not free. In opposite direction, there is a huge number of specific-purpose middleware solutions, precisely for IoT demands. Sensing data ingestion services and interoperable ways to manage IoT devices became dominant in the literature in the last decade.

Finally, collective operations, like reduce, prefix-sum and gather, are commonly used by general-purpose code while implementing tasks, but they are not mentioned in the middleware general-purpose literature. Even Spark map-reduce solution Zaharia et al. (2010) does not implement a RESTful version to implement interoperable mappers and reducers.

The Any.JS RESTful general-purpose computing middleware is an alternative also designed for elastic containerized cloud environments and capable to expose processing and storage services via a RESTful interoperable alternative, but without the previous mentioned limitations, i.e., with lock/unlock, iterator, register and observer services, but also with support for remote/asynchronous method calls with complex JSON parameters and not only String and primitive parameter types. The Any.JS instantiates/stores objects and calls methods implemented in Java, Python and JavaScript programming languages via its RESTful API.

Chapter 4

Development

In this chapter, it is detailed the Any.JS architecture, this way how its components are organized and how they interact. Besides the architecture, an example of how we can use the Any.JS services are presented at the end of the chapter.

4.1 Architecture

The Any.JS architecture is organized into layers, precisely into seven layers as Figure 4.1 illustrates.

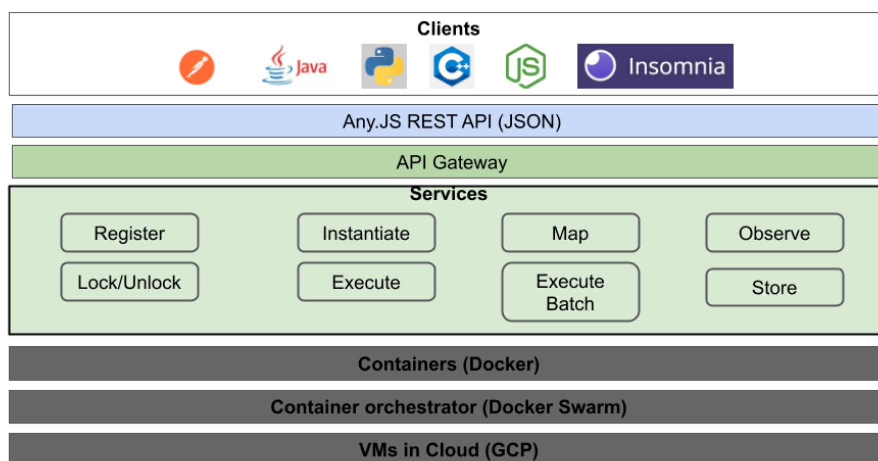

Figure 4.1: Any.JS architecture

4.1.1 The Virtual Machine (VM) Layer

The VM represents the first layer, this way the lowest abstraction level of the Any.JS system where the OS type, network configuration, memory capacity, number of cores on each virtual CPU or GPU and disk technology are applied. The Cloud environment turns the deploy and the maintainability of the systems a bit easier due to its elastic property. The Google Cloud Platform (GCP) and the AWS Platform are the two biggest players nowadays in Cloud Computing industry, so we decided to operate using GCP, but the Any.JS is compatible with AWS infrastructure and any other public Cloud Computing platform.

The Any.JS needs to be executed on Linux and requires at least a cluster composed of VMs with 2GB of RAM and 30GB of disk space minimum, these settings may change depending on the volume of jobs assigned to Any.JS. In summary, we just need to configure an elastic cluster environment, composed of VMs of any size.

4.1.2 Container Orchestrator

The second layer is responsible for the container orchestration, where services like run a task, precisely the gateway routes, cluster deployment, on demand cluster resize, services migration from one container to other and issues related with network are covered. There are two leaders nowadays: Docker Swarm Naik (2016) and Kubernetes Hightower et al. (2017). We decided to use Docker Swarm to scale the Any.JS services due to its integration with Docker containers and its simplicity when compared with Kubernetes.

To simplify all containers management, we decided to use Portainer Portainer (2021). It enables to manage the global cluster state, including delete images and volumes, track the memory/CPU usage on each container, and observe the microservices and VMs of the cluster in a more practical and friendly way. It is deployed with Any.JS, so transparent for the programmer. It is available through the IP address and port 9000. All these software dependencies and Docker cluster tuning issues are configured via Docker Swarm configuration file, so Any.JS has its own Docker Swarm configuration file.

4.1.3 Containers

Once we have deployed the container orchestrator, its necessary to deploy the containers. We have chosen the Docker container technology Merkel (2014) due to its popularity

and simplicity. Besides that, Docker Swarm and Docker containers are build from the same company, thus with high compatibility.

Figure 4.2 details the idea inside a container in the Any.JS system. Basically, we have processing containers and storage containers. On each container we have an image previously mounted and hosted in a public image repository (DockerHub - [1]), so we have Any.JS API services implementations containers for the programming languages Java, Python and JavaScript. Besides that, Any.JS has the observer and MongoDB Chodorow (2013) images. The observability service can be performed via RESTful API or using directly the Mongo DB client to be notified outside Any.JS. Processing and store service containers can be grouped into two containers type that can be replicated over the deployed VMs and managed by the Docker Swarm, as explained previously. The kernel of Any.JS is implemented in JavaScript, thus even the Java and Python containers have some of the JavaScript services, like Map, Store and Lock/Unlock.

The remaining services presented in Figure 4.1 are designed according to a specific programming language, but all of JavaScript or Python or Java services are described as JSON objects and performed via RESTful API. The programming language is part of a REST service URL to execute tasks and instantiate objects, since both service types are programming language dependable. The KrakenD container represents the Any.JS API gateway, thus implementing all services routes.

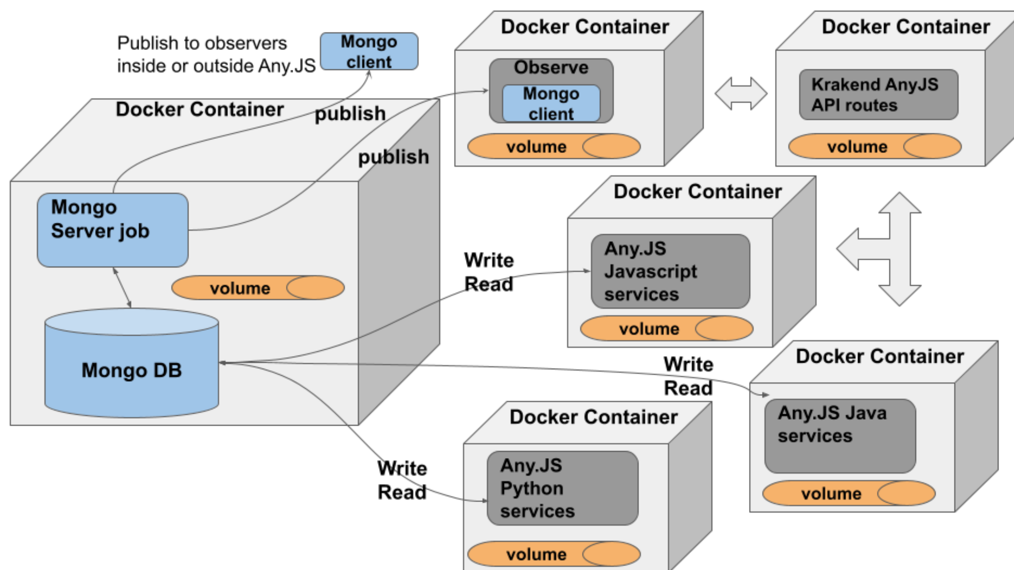

Figure 4.2: Any.JS container types

[1]DockerHub - <https://hub.docker.com/>

The MongoDB storage system is responsible for publishing updates to components named Observe (explained later in this chapter). These updates occur on MongoDB and using a collection concept. A collection is a set of documents and each document is a set of entries. MongoDB is well designed for JSON format and it stores collections of documents. When a collection is updated, MongoDB notifies its clients, being them an internal Any.JS container or a MongoDB client outside Any.JS cluster. This way, we created a way for the Any.JS users to observe collections changing in a simple manner. Most of Any.JS POST API requests creates a collection or document, so when the user stores a JSON object or when he or she instantiates or executes a task or perform an algorithm registration request, a collection or document is created and notifications are submitted for consumers interested on these events.

From the programmers perspective, the container configuration of the Any.JS is performed as follows: First, it is necessary to have a Docker Swarm cluster, configured according to any existing Web tutorial. On the manager machine of the cluster, you must install the Docker Compose configuration file. Next, you need to clone the source code from Any.JS[2] and, as root user, you have to run the file *run.sh*, located on root directory. This file deploys the Any.JS system on Docker Swarm.

To improve the performance of the microservices running on the Any.JS cluster, it is necessary to scale the server workers and for that there are two ways: you can use the Docker Command Line Interface (CLI) or you can edit the configuration file. We illustrate the option using the configuration file. Before you run the *run.sh* file, you must set the number of replicas of the server worker on the $docker - compose.yml$ file, located on root directory, precisely at line 16 of such file. The default value are three replicas, but this number must be changed according to your tasks requirements and cluster resources. After this configuration setup, you just need to run the *run.sh* file.

4.1.4 Services

There are eight services types and they are: Register, Execute, Execute Batch, Store, Instantiate, Lock and Unlock, Map and Observe. Each service is composed at least of four microservices: one to insert or run the service, one to search for metadata or content about the service previously executed, one to delete the service from Any.JS and one to update information about the service (completely or partially). An update requires very often the re-execution of the service.

[2] <https://github.com/lucasurzedo/AnyJS>

The service types details:

- Register: Responsible for upload all JavaScript or Java or Python code to the cluster. This process must be done at least once because many other service types, like execute and instantiate, use these codes to perform their jobs. The user just inform the location via URL of the code and Any.JS performs the download.

- Execute: It will execute a method of a class in an asynchronous and decentralized way. For that, the programmer must, through a JSON, pass to the API the parameters to execute the method, the previous registered module and the name of the method. The method parameters can be simple ones or complex JSON objects.

- Execute Batch: It will execute the same method of a class multiple times with different parameters. All executions are asynchronous and decentralized. Internally, Any.JS receives all runs and distribute them among all Any.JS containers available in the cluster. Each container can receive multiple method runs because there is a bag of tasks on each Any.JS running container to handle them. In the programmer perspective, it is necessary to mount a JSON and call the API just once with such JSON to perform multiple method calls. The method parameters can be simple ones or complex JSON objects. Since Execute Batch runs a method multiple times, there is a set of set of parameters in the JSON, one set per method run. The results of a batch can be obtained asynchronously via Observer, which means that each task result can be obtained individually, avoiding all batch method runs conclusion.

- Store: Responsible for storing an object already instantiated by the user. The object must be serialized to Binary JSON (BSON) format.

- Instantiate: Responsible for instantiating an object and storing it in the cluster, this way, the user must specify the constructor name, the previous registered module and the parameters (similar to *Execute* service).

- Lock and Unlock: Performs secure access to a variable previous created via *Instantiate* or *Store* services, or to a map entry previous created and populated via *Map* services. Both *Lock* and *Unlock* primitives require the name of the variable or the key of a map entry. They guarantee concurrent accesses of resources using a simple mutual exclusion solution. Any.JS uses a FIFO access order internally. Besides that, *Lock* and *Observe* are the unique synchronous services of the entire RESTful API.

- Map: Represents a transparent JavaScript distributed map in the API, thus all methods available in many map data structures of many programming languages are made available for use in the API, including the *iterator*, *putALL* and so many others.

- Observe: Represents a catalog of collections of Any.JS and a publish/subscribe mechanism. Applications can subscribe to these collections if the user wants to be notified of a service state change, being it a task execution completion or a variable storage or any other API option presented in Figure 4.1.

4.1.5 API Gateway

This layer represents a way to simplify the URL syntax, avoiding to expose internal design or implementation issues. For instance, to execute a task the programmer must perform a POST API call like this - '*IPaddress/api/anyJS/v1/execute/python/*'. Internally, the Any.JS system can call any microservice implementation, including third party alternatives. It is an elegant way to isolate user demands from technical issues, improving maintainability, extensibility and testability.

We have selected the KrakenD gateway KrakenD (2021) and it is configured by the container orchestrator, so the programmer does not need extra configuration demands.

4.1.6 Any.JS REST API

There are eight services types on Figure 4.1 and all of them are called following two simple rules. The rules for URL syntax are:

- *rule one*: All services have their name on the URL.
  - '*IPaddress/api/anyJS/v1/registry/*'
  - '*IPaddress/api/anyJS/v1/map/*'
  - '*IPaddress/api/anyJS/v1/storage/*'
  - '*IPaddress/api/anyJS/v1/instance/*'
  - '*IPaddress/api/anyJS/v1/task/*'
  - '*IPaddress/api/anyJS/v1/observer/*'

- *rule two*: The URL suffix can contain the programming language name. Ex. *execute*, *executebatch* and *instantiate* services have the following URLs:
  - '*IPaddress/api/anyJS/v1/instance/java/*'
  - '*IPaddress/api/anyJS/v1/instance/javascript/*'
  - '*IPaddress/api/anyJS/v1/task/python/*'
  - '*IPaddress/api/anyJS/v1/taskbatch/java/*'

The REST API calls are composed of JSON objects on their requests and responses. There are POST, GET, PUT, PATCH and DELETE HTTP requests. Each service type (Ex. execute or store) is composed of at least four microservices, as mentioned before, representing the Create, Retrieve, Update and Delete operations (CRUD) for each service type. The POST request is used to proceed an operation, i.e., a method execution or a object instantiation. Most of POST requests are asynchronous, this way the API caller receives an acknowledgment, indicating the URL to obtain the final result forward. Only Lock and Observe services are synchronous due to their characteristics. Each service type has at least one GET HTTP request to search for an existing map entry, a task result or any other service metadata information. Besides that, all service types have a DELETE and PUT/PATCH requests to enable all CRUD operations. The GET request is synchronous because it needs some result from the back-end and not only an acknowledgment. The GET request just obtain the result or the information that the service is under processing, thus GET requests do not wait services conclusion.

This work is not a tutorial about Any.JS API microservices, thus we decided to write the API using a specification. We decided to adopt the Open API Initiative API (2021). The server used to code and host the Any.JS system API is Swagger Hub Hub (2021). Register REST CRUD services, as well as the other API service types illustrated in Figure 4.1 are detailed at - <https://app.swaggerhub.com/apis-docs/lucasurzedo/AnyJS/1.0.0>.

There are three ways to wait for an API call: you can use the acknowledgment message with the URL to get the final result and proceed successive HTTP requests of GET type until a valid result is returned. There is a second alternative using a REST API call to observe the conclusion of a service call submitted previously - IPaddress/api/anyJS/v1/observe/collectionName/. The observe service is implemented using a library that creates a connection between MongoDB and the Web application framework, both deployed in different containers, thus notifications of collections updates are

performed without blocking the cluster with successive results checks. The third way is using the MongoDB client directly to observe Any.JS collections updates.

The Register API service summary:

- method POST: IPaddress/api/anyJS/v1/registry - creates a register with the URLs to download JavaScript or Java or Python executable modules with dependencies if necessary;
- method PUT: IPaddress/api/anyJS/v1/registry/{codeName} - updates the entire registered module;
- method GET: IPaddress/api/anyJS/v1/registry/{codeName} - retrieves a specific registered module metadata.
- method DELETE: IPaddress/api/anyJS/v1/registry/{codeName} - deletes a specific registered module stored in Any.JS.

The Execute API service summary:

- method POST: IPaddress/api/anyJS/v1/task/javascript/ - executes a specific method of a JavaScript previous registered module;
- method POST: IPaddress/api/anyJS/v1/task/java/ - executes a specific method of a Java previous registered module;
- method POST: IPaddress/api/anyJS/v1/task/python/ - executes a specific method of a Python previous registered module;
- method PUT: IPaddress/api/anyJS/v1/task/javascript/ - re-executes a specific method of a JavaScript previous registered module;
- method PUT: IPaddress/api/anyJS/v1/task/java/ - re-executes a specific method of a Java previous registered module;
- method PUT: IPaddress/api/anyJS/v1/task/python/ - re-executes a specific method of a Python previous registered module;
- method GET: IPaddress/api/anyJS/v1/task/{taskName} - retrieves all task executions stored in Any.JS;

- method GET: IPaddress/api/anyJS/v1/task/{taskName}/{executionName} - retrieves a specific task execution stored in Any.JS;
- method DELETE: IPaddress/api/anyJS/v1/task/{taskName} - deletes all task executions of a specific task name in Any.JS;
- method DELETE: IPaddress/api/anyJS/v1/task/{taskName}/{executionName} - deletes a specific task execution in Any.JS;

The Execute Batch API service summary:

- method POST: IPaddress/api/anyJS/v1/task/batch/javascript/ - executes a specific method of a JavaScript previous registered module multiple times, i.e., with different parameters in each task run;
- method POST: IPaddress/api/anyJS/v1/task/batch/java/ - executes a specific method of a Java previous registered module multiple times, i.e., with different parameters in each task run;
- method POST: IPaddress/api/anyJS/v1/task/batch/python/ - executes a specific method of a Python previous registered module multiple times, i.e., with different parameters in each task run;
- method PUT: IPaddress/api/anyJS/v1/task/batch/javascript/ - re-executes a specific and registered batch execution service call with different parameters if necessary;
- method PUT: IPaddress/api/anyJS/v1/task/batch/java/ - re-executes a specific and registered batch execution service call with different parameters if necessary;
- method PUT: IPaddress/api/anyJS/v1/task/batch/python/ - re-executes a specific and registered batch execution service call with different parameters if necessary;
- method GET: IPaddress/api/anyJS/v1/task/batch/{taskCollectionName} - retrieves all batch executions stored in Any.JS, where each batch execution has several task executions;
- method GET: IPaddress/api/anyJS/v1/task/batch/{taskCollectionName}/{executionName} - retrieves a specific batch execution stored in Any.JS, which includes its tasks executions metadata;

- method DELETE: IPaddress/api/anyJS/v1/task/batch/{taskName} - deletes all batch executions of a specific task name in Any.JS;
- method DELETE: IPaddress/api/anyJS/v1/task/batch/{taskName}/{executionName} - deletes a specific batch execution in Any.JS, including its tasks executions metadata;

The Store API service summary:

- method POST: IPaddress/api/anyJS/v1/storage/ - store a specific binary object that has already been instantiated;
- method PUT: IPaddress/api/anyJS/v1/storage/ - update a specific binary object that has already been stored;
- method GET: IPaddress/api/anyJS/v1/storage/{codeName} - retrieves all stored objects metadata of a code name;
- method GET: IPaddress/api/anyJS/v1/storage/{codeName}/{objectName} - retrieves a specific stored object;
- method DELETE: IPaddress/api/anyJS/v1/storage/{codeName}/{objectName} - deletes a specific stored object;
- method DELETE: IPaddress/api/anyJS/v1/storage/{codeName} - deletes all stored objects of a specific code name in Any.JS;

The Instantiate API service summary:

- method POST: IPaddress/api/anyJS/v1/instance/javascript/ - instantiates and stores a specific JavaScript object using a previous registered module;
- method POST: IPaddress/api/anyJS/v1/instance/java/ - instantiates and stores a specific Java object using a previous registered module;
- method POST: IPaddress/api/anyJS/v1/instance/python/ - instantiates and stores a specific Python object using a previous registered module;
- method PUT: IPaddress/api/anyJS/v1/instance/javascript/ - re-instantiates and stores a specific JavaScript object using a previous registered module;

- method PUT: IPaddress/api/anyJS/v1/instance/java/ - re-instantiates and stores a specific Java object using a previous registered module;
- method PUT: IPaddress/api/anyJS/v1/instance/python/ - re-instantiates and stores a specific Python object using a previous registered module;
- method GET: IPaddress/api/anyJS/v1/instance/{codeName} - retrieves all instantiated objects of a specific code name;
- method GET: IPaddress/api/anyJS/v1/instance/{codeName}/{objectName} - retrieves a specific instantiated object;
- method DELETE: IPaddress/api/anyJS/v1/instance/{codeName}/{objectName} - deletes a specific instantiated object;
- method DELETE: IPaddress/api/anyJS/v1/instance/{codeName} - deletes all instantiated objects of a specific code name;

The Map API service summary:

- method POST: IPaddress/api/anyJS/v1/map/ - creates an empty map;
- method POST: IPaddress/api/anyJS/v1/map/elements/ - creates map with several entries;
- method POST: IPaddress/api/anyJS/v1/map/entry/ - inserts a map element with a specified key and a value into a previous created map;
- method POST: IPaddress/api/anyJS/v1/map/forEach/javascript/ - executes a previous registered JavaScript module once per each key/value entry of the map;
- method POST: IPaddress/api/anyJS/v1/map/forEach/java/ - executes a previous registered Java module once per each key/value entry of the map;
- method POST: IPaddress/api/anyJS/v1/map/forEach/python/ - executes a previous registered Python module once per each key/value entry of the map;
- method PATCH: IPaddress/api/anyJS/v1/map/entry/ - updates a specific map entry with a new key-value pair;
- method PUT: IPaddress/api/anyJS/v1/map/elements/ - updates all map entries;

- method GET: IPaddress/api/anyJS/v1/map/entry/{mapName}/{key} - retrieves a specified element from a map;
- method GET: IPaddress/api/anyJS/v1/map/elements/{mapName} - retrieves all entries of a map;
- method GET: IPaddress/api/anyJS/v1/map/has/{mapName}/{key} - returns a boolean indicating whether an entry with the specified key exists or not;
- method GET: IPaddress/api/anyJS/v1/map/keys/{mapName} - retrieves the keys stored into a map;
- method GET: IPaddress/api/anyJS/v1/map/values/{mapName} - retrieves the values stored into a map;
- method DELETE: IPaddress/api/anyJS/v1/map/{mapName}/{key} - deletes a specified map entry;
- method DELETE: IPaddress/api/anyJS/v1/map/clear/{mapName} - deletes all elements from a map;
- method DELETE: IPaddress/api/anyJS/v1/map/{mapName} - deletes all elements from a map and the map itself;

The Lock and Unlock API service summary:

- method POST: IPaddress/api/anyJS/v1/sync/obj/ - creates a lock call for a previous stored object. It is a synchronous call, thus the caller waits until the lock succeeds or until a timeout occur;
- method POST: IPaddress/api/anyJS/v1/sync/map/ - creates a lock call for a previous stored map entry of a previous created map. It is a synchronous call, thus the caller waits until the lock succeeds or a timeout occur;
- method POST: IPaddress/api/anyJS/v1/unsync/obj/ - submits a new object value and unlocks a previous locked object. Cannot perform unlock operations to not locked objects or before the lock ordering;
- method POST: IPaddress/api/anyJS/v1/unsync/map/ - submits a new map entry and unlocks a previous locked map entry. Cannot perform unlock operations to not locked map entries or before the lock ordering;

- method GET: IPaddress/api/anyJS/v1/sync/obj/{objName} - retrieves all lock calls metadata from an object;
- method GET: IPaddress/api/anyJS/v1/sync/map/{mapName}/{key} - retrieves all lock calls metadata from a map entry;
- method DELETE: IPaddress/api/anyJS/v1/sync/obj/{objName}/id/{identifier} - deletes a lock call from an object queue of locks;
- method DELETE: IPaddress/api/anyJS/v1/sync/map/{mapName}/{key}/id/{identifier} - deletes a lock call from a map entry queue of locks;

The Observe API service summary:

- method POST: IPaddress/api/anyJS/v1/observer/ - observes any previous API call, i.e., we can observe a task execution or an object instantiation or even a map creation or its entries insertions. It is a synchronous call, thus the caller waits until the operation finishes or a timeout occur;
- method DELETE: IPaddress/api/anyJS/v1/observer/{operationName} - deletes an existing observation;
- method GET: IPaddress/api/anyJS/v1/observer/{operationName} - retrieves all observations metadata from a previous submitted operation;

4.1.7 Any.JS Clients

The last layer of Figure 4.1 is represented by several clients in different programming languages or tools, like Insomnia Insomnia (2021), to test REST APIs. This layer represents a way to simplify even more the Any.JS API microservices presented on previous section. Each independent company or volunteer can design and implement its own clients with different microservices composition strategies.

We have developed one client in JavaScript and it is used on the next section to develop the examples. Such a JavaScript client interfaces documentation and code are available at (repository - <https://github.com/lucasurzedo/AnyJS-client>). We omit its details in this section, since on Section 4.2 we have explained its utilization during the example explanation.

Note that it is possible to submit Java or Python code to Any.JS using such JavaScript Any.JS client. We have performed this logic to evaluate Hazelcast Java code, this way the same code is tested using Hazelcast alone and Any.JS with Java support. This kind of feature highlights the usage possibilities for Any.JS.

4.2 The Any.JS API usage example

This example is useful to illustrate the Any.JS API services usage with the Any.JS JavaScript client facilities [3]. First the Any.JS object is created from a specific host (Lines 1 to 4).

```
1 import AnyJSClient from '../client/AnyJSClient.js';
2
3 const host = 'http://34.148.224.114'
4 const anyJSClient = new AnyJSClient(host);
```

Next, there is how to register an existing code in Any.JS, as well as how to get information about it (Lines 7 to 9).

```
7 await anyJSClient.registerCode('fatorial',
  ↪ 'https://pastebin.com/raw/2YNiBLVv');
8 await anyJSClient.registerCode('pearson',
  ↪ 'https://pastebin.com/raw/NYjjcUyF');
9 await anyJSClient.getCode('pearson');
```

Next, the user can run a task, precisely the *fatorial* algorithm (Line 11), since such code has being previously registered. The user must inform the previous registered module, a task run label, the method to be called and its arguments. In Line 12, the user decided to obtain metadata about the task running or the task result itself. This way, the Any.JS *getRun* method does not wait for a task conclusion, but it always returns a result. Line 13 illustrates how to create a synchronization barrier in the user application. The Observer API service is responsible for that, since it informs when a task finishes or when an instantiation finishes or when any other API call result change. Finally, the user can delete a single task run or all task runs (Lines 14 and 15, respectively).

[3] avaliable at - <https://github.com/lucasurzedo/AnyJS-client/tree/main/examples>

```
11 await anyJSClient.run('fatorial', 'fatorialRun1', 'calcular', [10]);
12 await anyJSClient.getRun('fatorial', 'fatorialRun1');
13 await anyJSClient.observe('fatorial', 'fatorialRun1');
14 await anyJSClient.deleteRun('fatorial', 'fatorialRun1');
15 await anyJSClient.deleteAllRuns('fatorial');
```

Any.JS has an API service to store an object, but it also has a service to instantiate an object remotely. A *Pearson* object is instantiated by the user and stored by Any.JS on Lines 17 and 18. Lines 19 and 20 are responsible for a thread-safe update operation, where a new object is stored in the cluster. The instantiate service is a bit different, since the user must insert also the *Pearson* constructor parameters (Line 22) - $['John','Doe',1000]$. There is also a thread-safe update operation to instantiate a new object remotely in the cluster (Line 25). The user can print instances contents in Any.JS, but the stored objects cannot be printed because the store service receives binary objects that could not be cast by Any.JS.

```
17 const pearson1 = new PersonExperiment('John', 'Doe', 1000);
18 await anyJSClient.store(pearson1, 'onePearson');
19 const pearson2 = new PersonExperiment('Katy', 'Doe', 5000);
20 await anyJSClient.updateObject(pearson2, 'onePearson');
21 await anyJSClient.deleteObject('onePearson');
22
23 await anyJSClient.instantiate('pearson', 'anotherPearson', ['Mary',
   ↪ 'Doe', 12]);
24 await anyJSClient.printInstance('anotherPearson');
25 await anyJSClient.updateInstance('pearson', 'anotherPearson', ['Mary',
   ↪ 'Doe', 1200]);
26 await anyJSClient.printInstance('anotherPearson');
27 await anyJSClient.deleteInstance('anotherPearson');
28 await anyJSClient.deleteCode('pearson');
```

The user can create an empty map and insert several entries on it (Lines 30 to 37), but in a non thread-safe manner. A single map entry (key-value) can also be inserted or updated (Lines 38 and 39, respectively). The single map entry insertion can be done unsafely (Line 38) or in a thread-safe manner (Line 39). The *forEach* API call (Line 40)

is useful to perform many *fatorial* runs using the values stored into *oneMap* Any.JS map. The outputs of the *fatorial* runs are stored as new values of the existing map entries. The user can update several map entries and not just one (Lines 41 to 49), but unsafely, as mentioned before. The user can get a single or all map entries (Lines 50 and 51) and print the map content (Line 52).

```
30 await anyJSClient.createMap('oneMap');
31 const entries = {
32         key1: 10,
33         key2: 20,
34         key3: 30,
35         key4: 40,
36     }
37 await anyJSClient.putEntries('oneMap', entries);
38 await anyJSClient.put('oneMap', 'key5', 50);
39 await anyJSClient.lockPutUnlock('oneMap', 'key1', 12);
40 await anyJSClient.forEach('oneMap', 'fatorial');
41 const newEntries = {
42         key1: 50,
43         key2: 60,
44         key3: 70,
45         key4: 80,
46             key5: 90,
47             key6: 100,
48     }
49 await anyJSClient.putEntries('oneMap', newEntries);
50 await anyJSClient.getEntries('oneMap');
51 await anyJSClient.getEntry('oneMap', 'key1');
52 await anyJSClient.printEntries('oneMap');
```

Finally, the map can be traversed using *hasNext* and *next* API methods, respectively (Lines 54 and 55). The user can clear a map or delete it (Lines 57 and 58).

```
54 while (await anyJSClient.hasNext('oneMap'))
55         await anyJSClient.next('oneMap');
```

```
56
57 await anyJSClient.clearMap('oneMap');
58 await anyJSClient.deleteMap('oneMap');
```

4.3 Any.JS example used against Ignite

This example presents the Any.JS code used in the experiments against Ignite, thus very similar with the Ignite Node.js thin client [4]. Both implementations insert, get and delete 2k key-value map entries over a cluster using binary objects and Any.JS also with an option to insert *Pearson* objects without serialization. In terms of Any.JS, these binary objects are binary JSONs or BSONs, for short.

The code has approximately 170 lines, thus we decided to show only the core API calls to illustrate both Any.JS API and its Javascript client possibilities. First, from Line 1 to Line 7 the constants and the Any.JS backend host are informed. Next, the class *Pearson* is presented (Lines 9 to 25). A *Pearson* object has *id*, *firstname*, *lastname* and *salary* fields. In the experiment, 2k *pearsons* were inserted into a distributed map and then obtained all entries from such map.

```
9  class Pearson {
10   constructor(firstName = null, lastName = null, salary = null) {
11     this.id = Pearson.generateId();
12     this.firstName = firstName;
13     this.lastName = lastName;
14     this.salary = salary;
15   }
16
17   static generateId() {
18     if (!Pearson.id) {
19       Pearson.id = 0;
20     }
21     const id = Pearson.id;
```

[4] available at <https://github.com/apache/ignite-nodejs-thin-client/blob/master/examples/CachePutGetExample.js>

```
22      Pearson.id++;
23      return id;
24    }
25  }
```

The *MapExp* class has four core methods: *setObjects*, *setBinaryObjects*, *getObjects* and *getBinaryObjects*. The *set* methods are responsible for insert *pearson* objects into a hash map called *pearsons*. The binary version converts such objects into BSONs, as mentioned before.

The *setObjects* method creates four *pearson* objects (Lines 54 to 57) and inserts them using the Any.JS API call *putEntries* (Lines 60 to 65). A set of four pearsons is inserted into a map per time during the experiment. In the experiments, we call *setObjects* multiple times to stress Any.JS. The similar *setBinaryObjects* method just converts the four pearsons into BSONs before put them into an Any.JS map via *putEntries* API call. The method *setBinaryObjects* is similar to *setObjects*, but it has a serialization of the four pearsons before a *putEntries* API call.

```
53    async setObjects() {
54      const pearson1 = new Pearson('John', 'Doe', 1000);
55      const pearson2 = new Pearson('Jane', 'Roe', 2000);
56      const pearson3 = new Pearson('Mary', 'Major', 1500);
57      const pearson4 = new Pearson('Richard', 'Miles', 800);
58
59
60      await anyJSClient.putEntries(MAP_NAME, [
61        { key: `pearson${this.objIndex++}`, value: pearson1 },
62        { key: `pearson${this.objIndex++}`, value: pearson2 },
63        { key: `pearson${this.objIndex++}`, value: pearson3 },
64        { key: `pearson${this.objIndex++}`, value: pearson4 },
65      ]);
66
67      console.log('Storing Pearson Objects...')
68    }
```

The method *getObjects* of class *MapExp* returns all map entries stored into the

cluster and for that several *getEntry* API calls are performed (Lines 96 to 100). The *getBinaryObjects* version is similar to *getObjects* method, requiring *pearson* objects deserializations.

```
92    async getObjects() {
93      console.log('Pearson Objects getAll:');
94
95      const promises = [];
96      for (let index = 0; index < this.objIndex; index++) {
97        promises.push(anyJSClient.getEntry(MAP_NAME,
       ↪  `pearson${index}`).then((pearson) => {
98          this.printPearson(pearson);
99        }));
100     }
101
102     await Promise.all(promises);
103   }
```

After the *MapExp* class code, we have the two experiments code: methods *startExp*1 and *startExp*2, where the first implements the serialized version of a map manipulation and the other a non-serialized version. We decided to explain just one because they are very similar.

In the first part of the *startExp*1 method we evaluated map entries insertions and retrievals and for that we performed several *setBinaryObjects* calls (Lines 128 to 136). As explained previously, each of this API call inserted four *pearson* objects into a map. A way to get all map entries is illustrated by Line 136.

```
122 async function startExp1() {
123   const startExecution = Date.now();
124
125   const mapExp = new MapExp(iterations);
126   await anyJSClient.createMap(MAP_NAME);
127
128   let promises = [];
129   for (let index = 0; index < iterations; index++) {
```

```
130      promises.push(mapExp.setBinaryObjects());
131    }
132
133    await Promise.all(promises);
134
135    console.log('Getting binary objects...');
136    await mapExp.getBinaryObjects();
137
138    await mapExp.clearMap();
139
140    const endExecution = Date.now();
141    console.log(`\nTime taken to putALL and getALL map binary entries =
       ↪  ${(endExecution - startExecution) / 1000} seconds`);
142  }
```

The entire code used to test map manipulations between Any.JS and Ignite is available in Any.JS Github repository.

4.4 Any.JS example used against Hazelcast

The code to be compared with Hazelcast version is organized into the *TaskExecuteExample* class (Lines 8 to 54). Basically, the methods *sendMultipleTasksJava*, *sendMultipleTasksJS*, *getTaskResults* and *deleteTask* are implemented to submit factorial tasks in Java and in Javascript, to get all these factorial runs results and finally a way to delete a task, including all its runs, respectively.

The method *sendMultipleTasksJava* (Lines 13 to 25) starts an observer to watch any modification in the collection named *factorial_java* (Line 17), which means any task result from the 50 submitted tasks. After that, several *executeCode* API calls are performed (Lines 18 to 22). As mentioned before, the *Factorial* task is executed 50 times in the cluster asynchronously and remote.

```
13    async sendMultipleTasksJava() {
14      let methodArgs = [];
15      let number = 16;
```

```
16
17      const observer = anyJSClient.observerCollection('task',
        ↪  'factorial_java', iterations);
18      for (let i = 0; i < iterations; i++) {
19        methodArgs = [number];
20
21        await anyJSClient.executeCode('factorial_java', `exec${i}`,
          ↪  'factorial', 'java', 'Factorial', [], methodArgs);
22      }
23
24      return observer;
25    }
```

In sequence, the user can *getTaskResults* (Lines 41 to 48), where the task name must be informed. The method waits all tasks conclusions with a *promise* call (Line 500). The tasks results are obtained via *getExecution* method (Line 45).

```
41    async getTaskResults(task) {
42      let promises = [];
43
44      for (let i = 0; i < iterations; i++) {
45        promises.push(anyJSClient.getExecution(task,
          ↪  `exec${i}`).then((result) => {
46          console.log(`Iteration-${i}: ${result}`);
47        }));
48      }
49
50      await Promise.all(promises);
51    }
```

After the *TaskExecuteExample* class code, we have the experiments running 50 times the factorial of element 16. The method *start*1 implements the Java experiment with Any.JS (Lines 59 to 72). The method *start*2 implements the Javascript experiment with Any.JS (Lines 76 to 89). Due to similarity, we present one of them, precisely the Java experiment.

```
59 async function start1() {
60   const startExecution = Date.now();
61 
62   const taskExecuteExample = new TaskExecuteExample();
63 
64   console.log(await taskExecuteExample.sendMultipleTasksJava());
65 
66   await taskExecuteExample.getTaskResults('factorial_java');
67 
68   await taskExecuteExample.deleteTask('factorial_java');
69 
70   const endExecution = Date.now();
71   console.log(`\nTime taken to execute = ${(endExecution -
   ↪  startExecution) / 1000} seconds`);
72 }
```

The entire code used to test task processing between Any.JS and Hazelcast is available in Any.JS Github repository [5].

[5]available at - <https://github.com/lucasurzedo/AnyJS-client/tree/main/examples>

Chapter 5

Experiments

This chapter presents the cluster setup where we executed the Any.JS system and the performed experiments. The Ignite, Hazelcast and Any.JS examples were evaluated using the same cluster setup and one middleware at the time.

5.1 VMs setup

The environment was configured in small clusters deployed in the GCP. They are composed of five VMs running in the same network, and each VM has its own external Internet Protocol (IP). The cluster configuration is presented in Table 5.10.

Table 5.1: VMs configuration

VM	OS	CPU Model	CPU Cores	System RAM	System Storage
1	Ubuntu 20.04	On Demand	2 vCPUs not shared	2GB	30GB
2	Ubuntu 20.04	On Demand	2 vCPUs not shared	2GB	30GB
3	Ubuntu 20.04	On Demand	2 vCPUs not shared	2GB	30GB
4	Ubuntu 20.04	On Demand	2 vCPUs not shared	2GB	30GB
5	Ubuntu 20.04	On Demand	2 vCPUs not shared	2GB	30GB

5.2 Containers setup

On every cluster we have three container setups - one for Any.JS, one for Hazelcast and a third for Ignite. Figure 5.1 illustrates the container setups. The Docker Swarm manages several containers for Any.JS and they are: Krakend gateway container with

several routes, MongoDB container for persistence demands and several Flask containers to receive requests and send responses in the Web. Krakend and MongoDB containers are deployed into Swarm master VM. Besides these containers, we can have as many container as the user wants by deploying Any.JS, Hazelcast and Ignite container types with all API services available and they represent the Swarm worker containers. A Flask Web server is also deployed on these worker containers, since Any.JS is a REST solution. For Ignite, the Swarm manages the head/master container into the master VM and Ignite workers as Swarm work containers. We have deployed 1, 2 and 4 Swarm worker containers of type Any.JS services or Hazelcast and Ignite workers, over the cluster with 4 VMs.

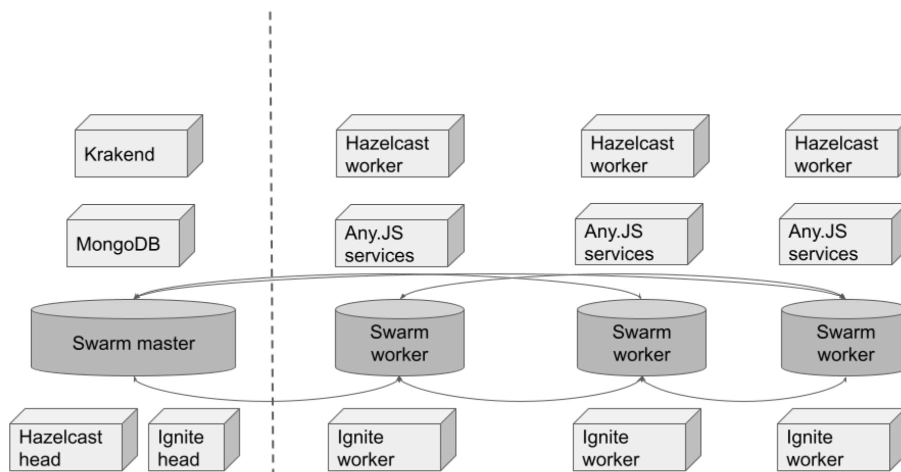

Figure 5.1: Any.JS, Hazelcast and Ignite container configurations

5.3 Metrics

We calculate two metrics in our experiments: runtime and CPU consumption. Each experiment were measured in terms of runtime and CPU consumption.

- Runtime: represents the period during which a computer program is executing. In our experiments, the runtimes are obtained in the client application and after many API calls.

- CPU usage: also called CPU footprint, represents the CPU consumption (zero to 100%) when the container is idle and running any program.

5.4 Any.JS versus Ignite

5.4.1 Any.JS Experimental Results

The code presented in Section 4.3 is used in this experiment. Figure **??** illustrates the Any.JS results using a cluster with 1 to 4 VMs, one for master Docker Swarm deploy and 1 to 4 others for its workers. Any.JS Docker image is deployed on every Swarm worker. We have deployed 1, 2 and 4 containers of Any.JS worker type.

The left most two tables represent both binary and *Pearson* insert and get operations runtimes. The two right most tables represent the CPU usage using 1 to 4 VMs and containers. We have performed 250, 500, 1k and 2k map operations. The first result is the marginal runtime difference (less than 10%) when we use binary or JSON object representations. Ignite, for instance, implements only binary option. In Any.JS the user can decide without loss in performance.

A second important result is that after 1k insertions and get all map entries even the cluster with 4 VMs and 4 containers became 100% used, which means the MongoDB operations (write and read concurrently the same JSON collection that represents the map). Note that, we have duplicated the number of map entries (1k to 2k), but the runtimes almost triplicated. The test with 2k map entries represents too many concurrent API calls even for the largest cluster configuration (4 VMs and 4 containers). The MongoDB is deployed in a single container in the same VM of Swarm master deployment, thus it represents a bottleneck. A sharding MongoDB deployment can attenuate this limitation and it is part of future Any.JS improvements.

Table 5.2: Any.JS Binary Objects runtimes with 1 to 4 VMs and containers

Binary Objects			
Any.JS	1	2	4
250	3.416 s	2.858 s	2.83 s
500	6.663 s	5.299 s	5.895 s
1000	14.441 s	12.683 s	15.604 s
2000	39.178 s	36.123 s	41.84 s

Table 5.3: Any.JS Binary Objects CPU usage with 1 to 4 VMs and containers

Any.JS	1	2	4
250	100%	100%	65.74%
500	100%	100%	78.14%
1000	100%	100%	98.32%
2000	100%	100%	100%

Table 5.4: Any.JS Objects runtimes with 1 to 4 VMs and containers

Objects			
Any.JS	1	2	4
250	2.769 s	2.398 s	2.574 s
500	6.753 s	4.75 s	6.549 s
1000	13.943 s	11.594 s	14.003 s
2000	38.403 s	35.711 s	39.557 s

Table 5.5: Any.JS Objects CPU usage with 1 to 4 VMs and containers

Any.JS	1	2	4
250	100%	89.78%	68.34%
500	100%	100%	83.37%
1000	100%	100%	100%
2000	100%	100%	100%

5.4.2 Ignite Experimental Results

The code present in Ignite repository - [1] was used and the two first changes we have performed are call 100 times the method *start* of *CachePutGetExample* class (line 180) and remove the *igniteClient.destroyCache* call (line 85), avoiding 100 delete cache data structure operations and increasing the Ignite original code utilization because of the 100 iterations of 4 persons inserted into a map per iteration. Instead, we have performed just one *igniteClient.destroyCache* call at the end of the code. Besides that, we have implemented an asynchronous way to call multiple times a manipulation of four pearsons that are inserted in a synchronous way. We have implemented the new version using Javascript promises.

Figure **??** illustrates two tables representing the synchronous and asynchronous Ignite experiment runtime results, respectively. The cluster is identical to the Any.JS one, with 1 to 4 VMs and containers. The first important information is that the asynchronous

[1] available - <https://github.com/apache/ignite-nodejs-thin-client/blob/master/examples/CachePutGetExample.js>

version is 2 to 3 times faster than the synchronous Ignite version and the difference tends to increase as the number of map entries to be inserted increase.

The second important result from Ignite experiments is that even with 2k map entries concurrently inserted in a map, the cluster with 1, 2 and 4 VMs and containers did not saturate their capacity to manipulate a single map, since the runtimes did not increase from 1 to 4 VMs/containers.

Table 5.6: Ignite synchronous with 1 to 4 VMs and containers

Binary Objects			
Ignite Sync	1	2	4
250	1.922 s	2.434 s	2.657 s
500	3.907 s	4.901 s	6.299 s
1000	6.489 s	8.406 s	10.16 s
2000	12.009 s	15.065 s	17.561 s

Table 5.7: Ignite asynchronous with 1 to 4 VMs and containers

Binary Objects			
Ignite Async	1	2	4
250	1.174 s	0.943 s	1.035 s
500	2.053 s	1.994 s	2.473 s
1000	2.933 s	3.054 s	3.241 s
2000	4.944 s	5.142 s	5.18 s

5.4.3 Comparative Analysis

In summary, Any.JS is around 3 times slower than Ignite synchronous and around 7 times slower when compared with Ignite asynchronous. There are many reasons for such differences: i) Any.JS is a RESTful solution with REST common overheads, like HTTP protocol, JSON and so forth; ii) Any.JS uses MongoDB to store all map entries, thus MongoDB becomes a bottleneck. A sharding deployment of MongoDB can attenuate the concurrence level; iii) Ignite runs a Javascript example and Any.JS runs a Javascript example using a Javascript Any.JS client that calls the Any.JS RESTful API. As we can see, there are overheads in this Any.JS experimental design. We can evaluate Any.JS directly using its RESTful API and using Insomnia or any other REST API test tool.

5.5 Any.JS versus Hazelcast

Figure **??** illustrates both Any.JS implementations (Java - the left most table and Javascript - the right most table). Besides that, in Figure **??** there are Hazelcast Java runtimes (the darker left most table). We have submitted the factorial of 17 to the Any.JS and Hazelcast clusters 50 times each. The same Factorial Java code was used by Hazelcast and Any.JS. A Javascript Factorial version was implemented to evaluate Any.JS portability. Hazelcast requires a manual code registration before execution and Any.JS automatic register Java, Python and Javascript classes.

Table 5.8: Any.JS Java runtimes

Factorial Java			
Any.JS	1	2	4
50	0.754 s	0.607 s	0.597 s

Table 5.9: Any.JS JavaScript runtimes

Factorial JavaScript			
Any.JS	1	2	4
50	2.828 s	2.594 s	2.694 s

Table 5.10: Hazelcast Java runtimes

Factorial Java			
Hazelcast	1	2	4
50	0.14 s	0.137 s	0.139 s

As we can see, the Java Any.JS version is faster than its Javascript counterpart. In summary, the cost of calling a Java library from a Javascript code compensates. O average, Any.JS Java is almost 6 times faster than the Any.JS Javascript.

The Hazelcast is, on average, 5 times faster than Any.JS Java version and as we increase the number of tasks submitted to the cluster the runtime differences also increase. There are many reasons for such differences:

- Any.JS is a RESTful solution with REST common overheads, like HTTP protocol, JSON and so forth;
- Any.JS supports Java, Python and Javascript programming languages and Hazelcast only Java for both frontend and backend;

- Any.JS uses MongoDB to store all tasks runs results and metadata, thus MongoDB becomes a bottleneck. A sharding deployment of MongoDB can attenuate the concurrence level;

- Any.JS encapsulates the MongoDB client, providing a REST API to receive notifications from collections updates. This task conclusion notification mechanism introduces overhead. The Any.JS Javascript client can use the MongoDB client directly to attenuate these overheads;

- To run 50 tasks in Any.JS there is the *executeBatch* API service, but it was not used because Hazelcast does not implement such optimization. The *executeBatch* reduces communications between the user and the Any.JS backend, as well as it reduces communications from each service running with MongoDB instance;

- Any.JS maintains the task results after VM faults and Hazelcast does not, so if a VM with a task result drops down such results are lost forever, even if Kubernetes starts another VM and another Hazelcast worker container. A task re-execution in this situation is mandatory for Hazelcast.

Chapter 6

Conclusion

In this work we present a middleware tool, named Any.JS, to reduce a bit more the existing gap of the RESTful APIs for general-purpose middlewares. Very often, specific programming languages are mandatory to complete the coding of some fundamental processing and storage services, thus interoperability is reduced when we do not use JSON or BSON objects via HTTP requests/responses.

The Any.JS system is designed to operate over cloud distributed environments, thus it can be deployed on different cluster configurations and sizes. It is a containerized solution, so migration, fault tolerance and an flexible deploy over different VMs are feasible. The Docker Swarm is responsible for the container orchestration, where, for instance, service discovery and service recovery are done transparently. Processing and storage containers run JavaScript, Java or Python jobs. An interoperable RESTful API exposes microservices for users to register compiled modules or classes, store BSON objects, instantiate JSON objects remotely, create and manage distribute map data structures, and run a single or multiple tasks asynchronously and remote.

The literature about general-purpose middeware tools did not implement register service to perform registering of classes in runtime. The execution of tasks use only String and primitive types, as well as sometimes a tool does not have the REST service to run tasks remotely and asynchronously. The instantiate useful service is not performed by the literature, which means we do not have a way to create a huge object only in the backend-side and not on the frontend-side and transferred for the backend via costly network communication. The Map data structure API is incomplete in terms of REST utilization, e.g., the iterator Map operator to traverse it, lock/unlock operations into a map entry, and the storage of complex objects without transforming them in binaries. The concept of

variables and not always a data structure collection entry is sometimes useful because not always our abstractions are collections of items, but the literature considers collections as a fundamental storage abstraction during the code development using Ignite or Hazelcast, for instance.

We have evaluated Any.JS against Ignite in terms of storage and against Hazelcast in terms of processing services. Ignite has a Javascript API and an example where four *Pearson* objects are inserted into a map and retrieved from it. We extended such example to produce several API calls to insert many map entries and after that a *getAll* operation is performed to obtain all *Pearsons* from the cluster. Ignite requires object serialization and Any.JS does not. In terms of runtime, Ignite was 3 to 5 times faster, depending if we insert all map entries synchronously or asynchronously. It represents a huge difference, so the REST overhead can be considerable, including the Any.JS architectural design to implement a REST support. Improvements mentioned in the Experiments Chapter are fundamental to reduce a bit more these runtime differences.

Hazelcast is a market-leader and it outperformed Any.JS in all 50 task execution scenarios, which means in all cluster and container configuration setups. On average, Hazelcast was 5 times faster than Any.JS to run 50 times the *Factorial* task. The Any.JS runs tasks written in Java or Python or Javascript, so its interoperability pays a high cost. Hazelcast, on the other hand, has only a Java client to communicate with its backend.

Many improvements in Any.JS must be done. Many new experiments must be conducted. The *executeBatch* must be tested against Hazelcast. The MongoDB must be deployed in sharding mode. The current Observer RESTful API service must be improved to avoid unnecessary delays. The Any.JS Python client must use MongoDB client directly to accelerate notifications receives. Larger number of tasks and tasks with different workloads should be evaluated in Chapter Experiments. Other objects and not only *Pearson* must be evaluated with Ignite and Any.JS. Finally, the map iterator experiments must be included in Chapter Experiments to show how Ignite and Any.JS iterate over the distributed map entries.

Bibliography

aaaa.

aaaa.

Al-Jaroodi, J. and Mohamed, N. (2012a). Middleware is still everywhere!!! *Concurrency and Computation: Practice and Experience*, 24(16):1919–1926.

Al-Jaroodi, J. and Mohamed, N. (2012b). Service-oriented middleware: A survey. *Journal of Network and Computer Applications*, 35(1):211–220.

Almeida, A. L. B., Cimino, L. d. S., de Resende, J. E. E., Silva, L. H. M., Rocha, S. Q. S., Gregorio, G. A., Paiva, G. S., Delabrida, S., Santos, H. G., de Carvalho, M. A. M., et al. (2019). A general-purpose distributed computing java middleware. *Concurrency and Computation: Practice and Experience*, 31(7):e4967.

API, O. (2021). Open api specification. Available at: <https://www.openapis.org/>.

Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., and Zavattaro, G. (2005). Choreography and orchestration: A synergic approach for system design. In *International Conference on Service-Oriented Computing*, pages 228–240. Springer.

Chodorow, K. (2013). *MongoDB: the definitive guide: powerful and scalable data storage.* " O'Reilly Media, Inc.".

Christie, M. A., Bhandar, A., Nakandala, S., Marru, S., Abeysinghe, E., Pamidighantam, S., and Pierce, M. E. (2020). Managing authentication and authorization in distributed science gateway middleware. *Future Generation Computer Systems*, 111:780–785.

Elkady, A. and Sobh, T. (2012). Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of Robotics*, 2012.

Farahzadi, A., Shams, P., Rezazadeh, J., and Farahbakhsh, R. (2018). Middleware technologies for cloud of things: a survey. *Digital Communications and Networks*, 4(3):176–188.

Goldberg, R. P. (1974). Survey of virtual machine research. *Computer*, 7(6):34–45.

Hightower, K., Burns, B., and Beda, J. (2017). *Kubernetes: up and running: dive into the future of infrastructure.* " O'Reilly Media, Inc.".

Hub, S. (2021). Swagger hub. Available at: <https://swagger.io/tools/swaggerhub/>.

Ibrahim, N. and Mouel, F. L. (2009). A survey on service composition middleware in pervasive environments. *arXiv preprint arXiv:0909.2183*.

Insomnia (2021). Insomnia collaborative api design editor. Available at: <https://insomnia.rest/>.

Issarny, V., Bouloukakis, G., Georgantas, N., and Billet, B. (2016). Revisiting service-oriented architecture for the iot: a middleware perspective. In *International conference on service-oriented computing*, pages 3–17. Springer.

Jahantigh, M. N., Rahmani, A. M., Navimipour, N. J., and Rezaee, A. (2020). Integration of internet of things and cloud computing: A systematic survey. *IET Commun.*, 14(2):165–176.

Kjær, K. E. (2007). A survey of context-aware middleware. In *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*, pages 148–155. Citeseer.

KrakenD (2021). Krakend. Available at: <https://www.krakend.io/>.

Königsberger., J. and Mitschang., B. (2018). R2sma - a middleware architecture to access legacy enterprise web services using lightweight rest apis. In *Proceedings of the 20th International Conference on Enterprise Information Systems - Volume 2: ICEIS,*, pages 704–711. INSTICC, SciTePress.

Larian, H., Larian, A., Sharifi, M., and Movahednejad, H. (2022). Towards web of things middleware: A systematic review. *arXiv preprint arXiv:2201.08456*.

Marpaung, J. A., Sain, M., and Lee, H.-J. (2013). Survey on middleware systems in cloud computing integration. In *2013 15th International Conference on Advanced Communications Technology (ICACT)*, pages 709–712. IEEE.

Mell, P., Grance, T., et al. (2011). The nist definition of cloud computing.

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2.

Mohamed, N. and Al-Jaroodi, J. (2011). A survey on service-oriented middleware for wireless sensor networks. *Service Oriented Computing and Applications*, 5(2):71–85.

Nadareishvili, I., Mitra, R., McLarty, M., and Amundsen, M. (2016). *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.".

Naik, N. (2016). Building a virtual system of systems using docker swarm in multiple clouds. In *2016 IEEE International Symposium on Systems Engineering (ISSE)*, pages 1–3. IEEE.

Ngu, A. H., Gutierrez, M., Metsis, V., Nepal, S., and Sheng, Q. Z. (2016). Iot middleware: A survey on issues and enabling technologies. *IEEE Internet of Things Journal*, 4(1):1–20.

Peltz, C. (2003). Web services orchestration and choreography. *Computer*, 36(10):46–52.

Pérez, H. and Gutiérrez, J. J. (2014). A survey on standards for real-time distribution middleware. *ACM Computing Surveys (CSUR)*, 46(4):1–39.

Portainer (2021). Portainer. Available at: <https://www.portainer.io/>.

Razzaque, M. A., Milojevic-Jevric, M., Palade, A., and Clarke, S. (2015). Middleware for internet of things: a survey. *IEEE Internet of things journal*, 3(1):70–95.

Seovic, A., Falco, M., and Peralta, P. (2010). *Oracle Coherence 3.5*. Packt Publishing Ltd.

Sliwinski, W., Kaczkowski, K., and Zadlo, W. (2019). Fault tolerant, scalable middleware services based on spring boot, rest, h2 and infinispan.

Taboada, G. L., Ramos, S., Expósito, R. R., Touriño, J., and Doallo, R. (2013). Java in the high performance computing arena: Research, practice and experience. *Science of Computer Programming*, 78(5):425–444.

Vahdat-Nejad, H. (2014). Context-aware middleware: A review. *Context in computing*, pages 83–96.

Veentjer, P. (2013). *Mastering Hazelcast*. Hazelcast.

Walker, S. M., Dearle, A., Norcross, S. J., Kirby, G. N. C., and McCarthy, A. J. (2010). Rafda: A policy-aware middleware supporting the flexible separation of application logic from distribution. *ArXiv*, abs/1006.3728.

Wang, M.-M., Cao, J.-N., Li, J., and Dasi, S. K. (2008). Middleware for wireless sensor networks: A survey. *Journal of computer science and technology*, 23(3):305–326.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., Stoica, I., et al. (2010). Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95.

Zheludkov, M., Isachenko, T., et al. (2017). *High Performance in-memory computing with Apache Ignite*. Lulu.com.