SANTA CATARINA STATE UNIVERSITY - UDESC
COLLEGE OF TECHNOLOGICAL SCIENCE - CCT
GRADUATE PROGRAM IN APPLIED COMPUTING - PPGCAP

MARCO ANTONIO MARQUES

EVENT2LEDGER: CONTAINER ALLOCATION AND DEALLOCATION TRACEABILITY USING BLOCKCHAIN

JOINVILLE

2021

MARCO ANTONIO MARQUES

EVENT2LEDGER: CONTAINER ALLOCATION AND DEALLOCATION TRACEABILITY USING BLOCKCHAIN

Master thesis presented to the Graduate Program in Applied Computing of the College of Technological Science from the Santa Catarina State University, as a partial requisite for receiving the Master's degree in Applied Computing.

Supervisor: Dr. Charles Christian Miers

JOINVILLE

2021

Marques, Marco Antonio
Event2ledger: Container allocation and deallocation traceability using blockchain/ Marco Antonio Marques. – 2021.
87 p.

Orientador: Dr. Charles Christian Miers
Dissertação (mestrado) – Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Computação Aplicada, Joinville, 2021.

1. docker. 2. container. 3. monitoring. 4. blockchain. I. Miers, Charles Christian. II. Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Computação Aplicada. III. Título.

Marco Antonio Marques

Event2ledger: Container allocation and deallocation traceability using blockchain

Master thesis presented to the Graduate Program in Applied Computing of the College of Technological Science from the Santa Catarina State University, as a partial requisite for receiving the **Master's degree in Applied Computing**.

Master Thesis Committee:

Prof. Dr. Charles Christian Miers
Santa Catarina State University
President / Advisor

Prof. Dr. Marcos Antônio Simplício Jr.
University of São Paulo
Board member

Prof. Dr. Rafael Rodrigues Obelheiro
Santa Catarina State University
Board member

Joinville, 25th October 2021

I dedicate this work to everyone who believes that education paves the way for a better society.

ACKNOWLEDGMENTS

I can’t begin the acknowledgments without mentioning my parents. Without them, I literally wouldn’t be here. But, in addition to life, they gave me all the love, attention, and support so that I could become a decent man (a gentleman as my mother says). Let’s try as far as possible.

I also thank Professor Charles C. Miers, an essential person in this step, who believed in me and opened the doors so that I could, 13 years after graduation, return to my studies. During the entire journey, he was present, always with important and precise guidance and advice which allowed me to plan and carry out the master’s degree.

Last but not least, I thank everyone at home: my wife (who is crazy to be with someone crazy like me), Pitty (my dog, who attended many of the classes and meetings that I think she also deserves a diploma), to Tutu (the best engineer I’ve ever met), to the Piriquito, fish, frogs, plants and other beings with whom I live on this crazy journey that is life.

ABSTRACT

In a cloud computing scenario, in which services are developed and offered to customers typically on an outsourced platform, monitoring is an essential aspect in optimizing scalability, service delivery, failure detection, charging for the resources used, among other aspects. Concerning virtualization environments in Docker containers, the collection and analysis of container life cycle events allow to measure the application usage time, essential information for performance evaluation, consumption of computational resources, and billing, among others. In this scenario, a cloud provider can offer a monitoring tool, or each actor involved can also implement their own solution. However, the first case demands confidence from the participants in the Provider's collection and storage processes. The implementation of different monitoring solutions can lead to possible divergences regarding to the collected events. Thus, this work presents a monitoring solution proposal that collects events from containers and promotes consensus and distributed storage among the actors that compose the environment. We developed Event2ledger, a solution which provides mechanisms to grant event data integrity, authenticity, availability, irreversibility, and auditing. The implementation of a prototype using the Hyperledger Fabric blockchain, with Docker containers orchestrated by Docker Swarm, allow us to show the capability and operability of Event2ledger.

Keywords: Docker, container, monitoring, blockchain.

RESUMO

Em um cenário de computação em nuvem, no qual serviços são desenvolvidos e oferecidos a clientes tipicamente em uma plataforma terceirizada, o monitoramento é um aspecto essencial na otimização da escalabilidade, distribuição dos serviços, detecção e prevenção de falhas, cobrança pelos recursos utilizados, dentre outros aspectos. Tratando-se especificamente de ambientes de virtualização em contêineres Docker, a coleta e análise dos eventos de ciclo de vida de contêineres permite mensurar o tempo de uso das aplicações, informação essencial para avaliação de desempenho, consumo de recursos computacionais e tarifação, dentre outros. Neste cenário, o provedor pode oferecer uma ferramenta de monitoramento, ou cada ator envolvido pode também implementar sua própria solução. Contudo, o primeiro caso demanda confiança pelos participantes na coleta e armazenamento dos dados por parte do provedor. Já a implementação de soluções de monitoramento distintas pode levar a possíveis divergências quanto aos eventos coletados. Nesta dissertação é apresentada uma solução de monitoramento a qual coleta eventos gerados pelos contêineres e fornece um consenso e armazenamento distribuído entre os atores envolvidos neste cenário. A solução Event2ledger foi desenvolvida para fornecer mecanismos que assegurema integridade dos dados de eventos gerados com recursos de integridade, autenticidade, irreversibilidade e auditoria. A implementação de um protótipo usando a blockchain Hyperledger Fabric, com contêineres Docker orquestrados por Docker Swarm, possibilitou comprovar a capacidade e operacionalidade do Event2ledger.

Keywords: Docker, contêiner, monitoração, blockchain.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ABCI Application BlockChain Interface
ACL Access Control List
AMQP Advanced Message Queuing Protocol
API Application Programming Interface
AWS Amazon Web Services
BFT Bizantine Fault Tolerant
CA Certificate Authority
CaaS Container as a Service
cgroups Control Groups
CLI Command Line Interface
CFT Crash Fault Tolerant
CSA Cloud Security Alliance
CU Coherent Unit
DLT Distributed Ledger Technology
DoS Denial of Service
DDoS Distributed Denial of Service
DPI Deep Packet Inspection
DPoS Delegated Proof of Stake
EC2 Amazon Elastic Cloud Computing
ENISA European Network and Information Security Agency
EVM Ethereum Virtual Machine
FaaS Function as a Service
FBFT Federated Byzantine Fault Tolerance
FTP File Transfer Protocol
HTTP Hypertext Transfer Protocoll
IaaS Infrastructure as a Service
IP Internet Protocol
IoT Internet of Things
JSON JavaScript Object Notation
JWT JSON Web Token
LabP2D Laboratory of Parallel and Distributed Processing
LBaaS Load Balance as a Service
LVM Logical Volume Manager

LXC LinuX Containers
LXD Linux container hypervisor
MitM Man-in-the-Middle
MAC Media Access Control
MSP Membership Service Provider
mTLS mutual Transport Layer Security
NaaS Networking as a Service
NIC Network Interface
NIDS Network Intrusion Detection Systems
NIST National Institute of Standards and Technology
OS operating system
P2P Peer-to-Peer
PaaS Platform as a Service
PBFT Practical Byzantine Fault Tolerance
PPGCAP Graduate Program in Applied Computing
PoET Proof of Elapsed Time
PoW Proof of Work
QoS Quality of Service
RBFT Redundant Byzantine Fault Tolerance
REST Representational State Transfer
RPC Remote Procedure Call
RSA Rivest-Shamir-Adleman
S-VM System Virtual Machine
SaaS Software as a Service
SDK Software Development Kit
SDN Software Defined Networking
SGX Software Guard Extension
SLA Service Level Agreement
SPIFFE Secure Production Identity Framework For Everyone
SQL Structured Query Language
SPF Single Point of Failure
SSH Secure Shell
STP Spanning Tree Protocol
TLS Transport Layer Security
TSDB Time Series Database
TPS Transactions Per Second
TEE Trusted Execution Environment

UDESC Universidade do Estado de Santa Catarina

VLAN Virtual Local Area Network

VM virtual machine

VMM Virtual Machine Manager

WAN Wide Area Network

CONTENTS

1 INTRODUCTION

The microservices-based computing architecture makes it easy to build scalable, high-performance applications as they can be fragmented into independent pieces for development, version control, and provisioning (JAMSHIDI, 2018). In this model (NEWMAN, 2015), containers are considered the standard for implementation in the cloud due to their speed, ease of allocation, scalable management, and resilience.

The development of cloud computing providers allow companies to offer on demand computing resources, platforms, and applications. This scenario allowed developers and partners to offer their software and products to customers, running them on cloud computing platforms made available by providers. In this context, monitoring the execution of the virtual environment is important so that actors can optimize the execution of applications in the environment, track failures, charge / evaluate the charge for the computational resources used, among other activities.

In general, the cloud computing provider provides a monitoring system which allows customers to follow up on the container execution environment. This possibility gives monitoring autonomy to the actors that can adapt it to their needs. However, it directly depends on the trust in the provider regarding the guarantee of the integrity of the collected data and the monitoring solution adopted. On the other hand, customers can implement an independent monitoring service (DAWADI; SHAKYA; PAUDYAL, 2017). However, this possibility also brings the risk of divergences between actors, regarding the information collected by their tools, resulting in different analyzes of the same scenario. Specifically considering container lifecycle events, divergences in the collection of start, suspension, and termination events can result in an inaccurate analysis of the applications running in the environment, as well as the pricing and usage of available computing resources.

Thus, the main objective of this research is the development of a solution which collects and ensures consensus among the participants regarding the life cycle events of applications and containers. The specific objectives, in turn, are: (i) realize the event collection from applications and containers in a non-intrusive way; (ii) guarantee the authenticity of the components of the proposed solution as well as the secure communication between them; (iii) guarantee the integrity of the collected data, aim consensus between actors as to the order of events; and (iv) store these events in a verifiable distributed repository to audit and track possible changes. The method used in this work consists of a referenced research carried out to develop the theoretical foundation, followed by applied research to verify the feasibility and functionality of the proposed

solution. This work's main contribution is the development of a distributed monitoring solution that implements authenticity and integrity mechanisms and that promotes consensus regarding the validation and ordering of events performed by the actors.

The work's organization is the following: Chapter 2 introduces the theoretical foundation, approaching how the concept of cloud computing combined with container virtualization provides a favorable environment for the microservices architecture model. Based on the theoretical foundation, the basis for identifying the proposed research problem and its motivations is established.

Chapter 3 presents and details the Functional Requirements and Non-Functional Requirements that guide the development of the work. The research method adopted is also presented, as the search, inclusion and exclusion criteria, academic search mechanisms used, and, finally, how the identified related works meet or do not meet the established FR. Then, the proposed solution for the identified research question is detailed, addressing individually how the FR will be attended. With the definition of the proposed solution, the chapter also details the attack model developed to assess the main vulnerabilities to which monitoring solutions are exposed.

After defining the solution, Chapter 4 presents the theoretical foundation specific to the implementation model adopted, addressing the adopted container and orchestrator mechanism, as well as the blockchain model and consensus mechanism used. Then, the chapter presents the proof of concept, covering the development, configuration, and implementation of the proposed solution and its components, as well as the evaluation of its functionality and feasibility in the testing environment. Finally, the proposed solution is evaluated against the attacks listed in the described attack model to verify the efficiency of the adopted defense mechanisms.

2 FUNDAMENTAL CONCEPTS

The development of new technologies in computing is often accompanied by disruptive processes that affect both organizations and society in general. Specifically in virtualization, technological evolution has made it possible to abstract hardware, making it possible to share it over networks. This possibility gave rise to new business models, with organizations providing computing resources on-demand, in addition to the development of a software architecture in which applications are composed of several independent services. However, new opportunities bring new challenges, and this computing environment, composed of different actors (*e.g.*, customers, cloud computing providers, and developers) and applications require complex and secure monitoring, which provides reliable information to ensure the best performance, use, and pricing of computational resources.

In order to understand the problem presented in this work, it is necessary to go over some relevant aspects related to virtualization, its characteristics, models, and tools. Thus, this chapter presents the cloud execution model, its characteristics, responsibilities, and roles, in addition to the service models and types of deployment (Section 2.1). Then, the container virtualization model is described as an alternative to the virtualization model using virtual machine (VM), with features that guarantee a greater granularity of applications (Section 2.2). This high granularity led to the need to develop and use tools that allow the management of computational resources shared by users, such as orchestrators. These tools are responsible for the implementation, management, monitoring, and load balancing between the various containers in the environment, and are described in Section 2.3.

The work also presents the flow of service calls, from the customer to the container in execution, and points out that, through the container mechanism executed on each host is possible to carry out the monitoring of the environment through the collection of life cycle events of the containers and applications in real-time. Considering that distributed applications are composed of several containers and that an environment can contain several applications, there is a complex scenario, involving a high number of computational resources and services in execution, and highly dependent on monitoring for the efficient use of these resources, correct pricing and performance analysis, described on Section 2.4.

After describing the applications and tools that compose the environment, a common usage scenario is presented, involving different actors and their perspectives and highlighting the differences regarding the monitoring objective for each of them.

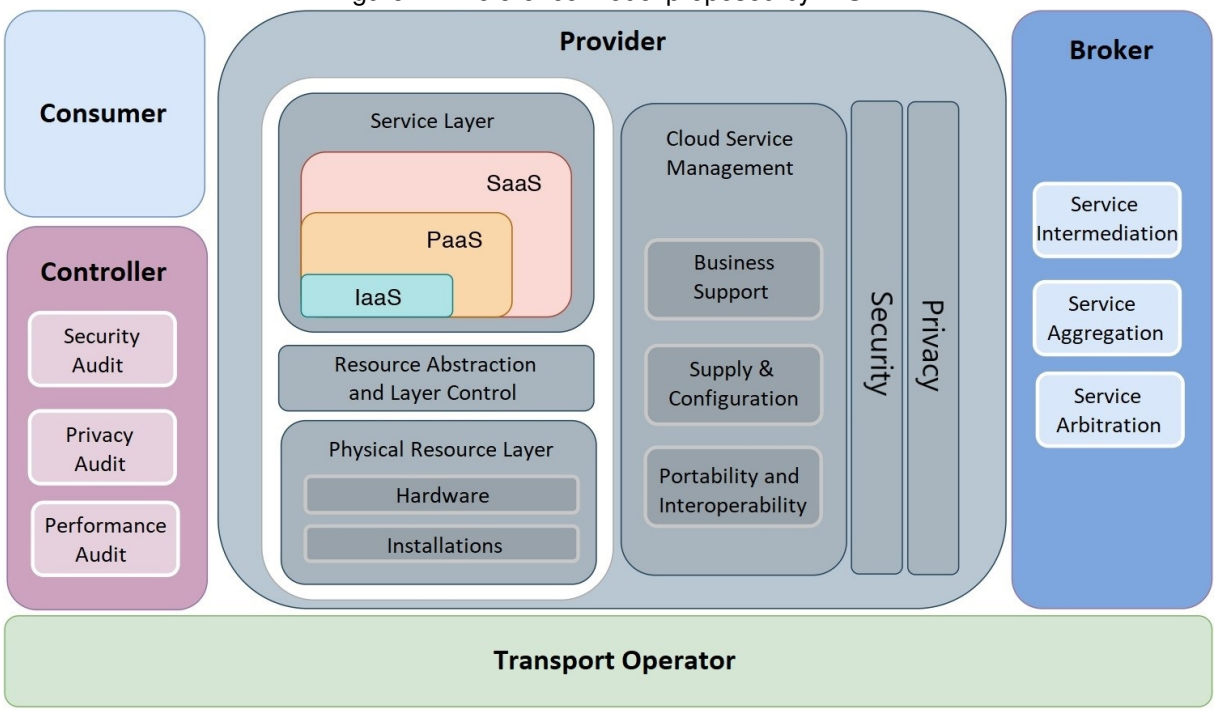
Finally, based on the concepts, actors, and needs to be involved, the problem (Section 2.5) and the partial considerations of this work (Section 2.6) are presented.

2.1 CLOUD COMPUTING

Cloud computing is a distributed computing architecture that involves different actors, services, and characteristics in making computing resources available on-demand via the network (*e.g.*, network, servers, storage, applications, and services) with minimum effort and interaction. The convergence of concepts such as pool of resources, virtualization, dynamic provisioning, and implementation on-demand, in addition to the delivery of resources via the Internet, make up the cloud computing architecture and allow a flexible approach in the implementation and scalability of applications (Tianfield, 2011).

As presented by (MELL; GRANCE, 2011), cloud computing allows network and on-demand access in a convenient and ubiquitous way to computational resources such as servers, storage, applications, and services. The architecture model proposed by National Institute of Standards and Technology (NIST) lists the characteristics of the cloud, the main service models, and the type of deployment. Figure 1 graphically represents the model described, based on the architecture proposed by NIST.

Figure 1 – Reference model proposed by NIST.



Source: (BOHN et al., 2011)

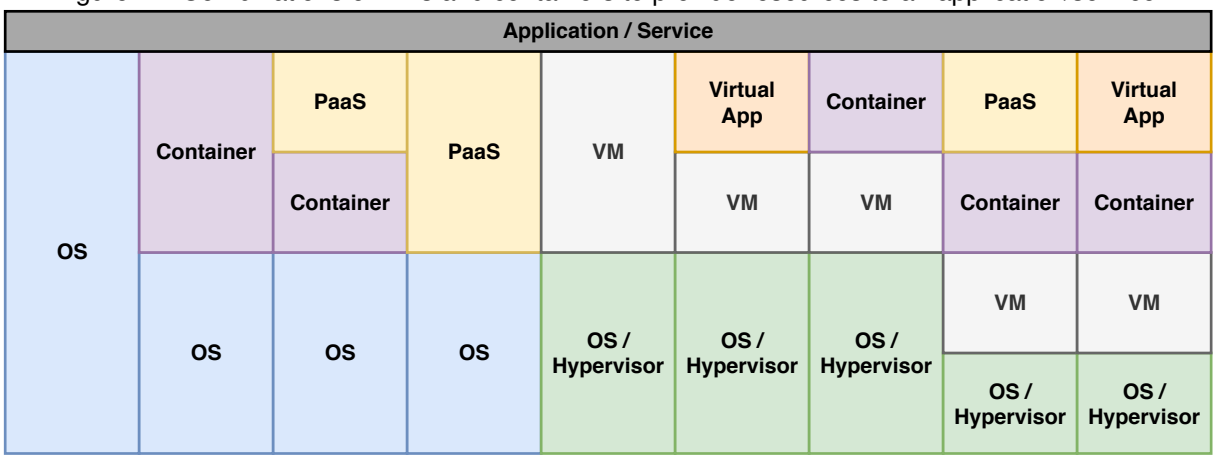
In the context of this work, it is relevant to highlight the role of three of the five actors presented: Consumer, Cloud Computing Provider, and Auditor (Figure 1). A consumer is a person or organization that uses the services offered by the cloud

computing provider, which, in turn, is the organization or entity responsible for providing the services. The auditor is the entity responsible for conducting independent audits of the services offered, in order to assess the use, performance, and security of the computational cloud.

According to (JADEJA; MODI, 2012), there are four main types of implementation of a computational cloud: Public Cloud, Private Cloud, Community Cloud, and Hybrid Cloud. In the Public Cloud, the consumer pays only for the time spent using the allocated resource, reducing their IT operation costs. However, because Public Clouds are more exposed, they are more susceptible to malicious attacks. In the Private Cloud, computational resources are not made available by the service server, but by the organization itself. A feature of this model is that it is easier to manage information security and less exposure to attacks. A Community Cloud consists of two or more Private Clouds that enter into a partnership (*e.g.*, joint-venture) to share their clouds with each other. Finally, there is the Hybrid Cloud, combining characteristics of the Public and Private models. In this model, a Private Cloud connects to one or more external cloud services, allowing the organization to meet its needs with the Private Cloud, but using complementations from the Public Cloud when necessary. Considering the present work, the computational cloud model to be used will be the Private Cloud model, in which the infrastructure is provisioned for the exclusive use of a pre-defined group of consumers.

In the context of computational cloud, containers and VMs are complementary technologies, which serve application implementations with different requirements. Figure 2 shows possible combinations of the use of these technologies.

Figure 2 – Combinations of VMs and containers to provide resources to an application/service.



Source: Adapted from (PANIZZON et al., 2019)

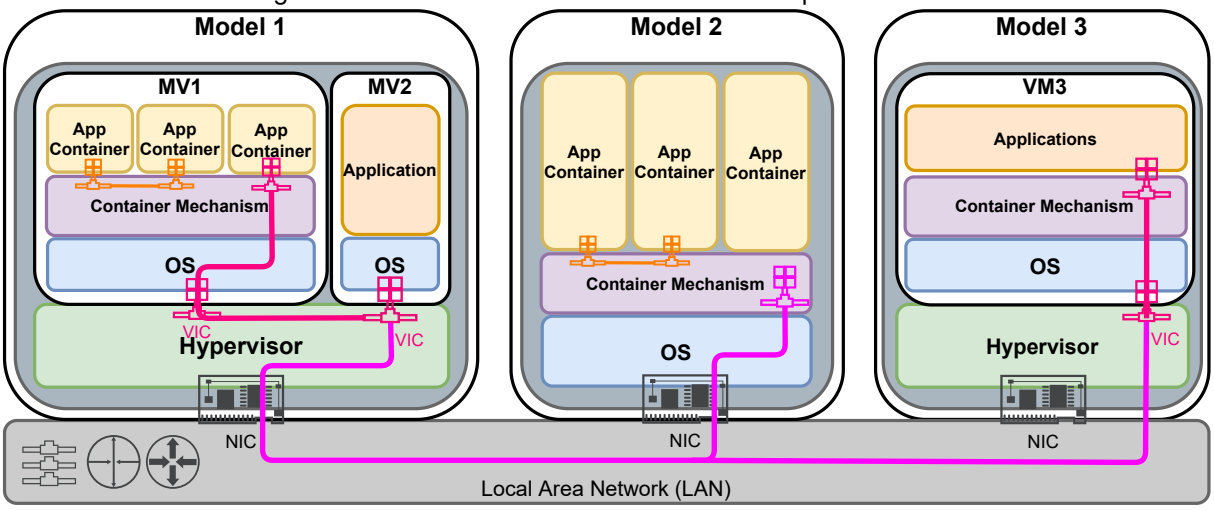
Each of the different proposals for combinations between containers and VMs presented in Figure 2 aims to meet the specific needs of the application and the environment to be implemented. Thus, if the objective is to implement a microservice

environment, one of the proposed combinations with container support is used. On the other hand, if the goal is to implement applications with a minimal structure, the common approach is operating system (OS) -Container, which has a superior performance to the model OS / Hypervisor-VM (PANIZZON et al., 2019). The present work has an objective to implant a container virtualization environment executed in VMs. There are several container virtualization technologies, among which Docker, Kubernetes and LinuX Containers (LXC) (RODRIGUEZ; BUYYA, 2018) stand out. Although different, these technologies have a set of characteristics common to all container virtualization models.

2.2 CONTAINER VIRTUALIZATION

Cloud-based systems involve several applications and services distributed across a group of hosts, which makes the process of orchestrating, creating, managing, and maintaining the infrastructure necessary for the implementation of the services complex (SYED; FERNANDEZ, 2018). Virtualization allows the abstraction of hardware, offering organizations new development models, use of packages or modules providing new services quickly, in addition to having features such as load balancing, scalability, and self-service (PORTNOY, 2016). Container-based virtualization is rapidly gaining in relevance. Its main application scope is still distributed applications, but there are already application use cases in Big Data among other workloads (DIAMANTI, 2019). Figure 3 presents the three main models for implementing containerized virtualization.

Figure 3 – Models of container virtualization implementation.

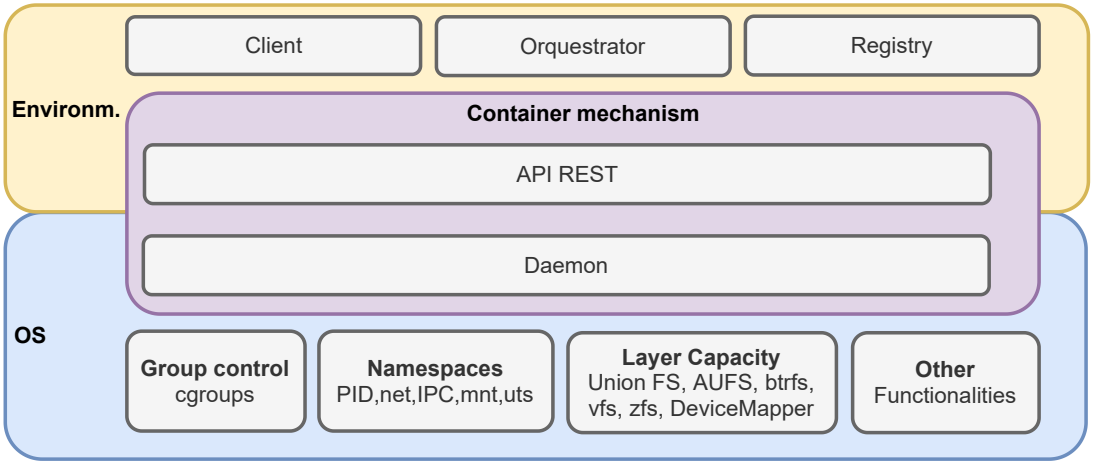


Source: (MIERS et al., 2019b)

Container virtualization can be implemented in VMs, according to Model 1, or directly on the physical machine (bare-metal), represented in Model 2. It is also possible to implement a model that runs the application in a containerized virtualiza-

tion environment running in VM (Model 3). The choice of the virtualization model to be implemented should consider characteristics such as performance and the desired isolation level. Regardless of the model adopted, they all have characteristics in common, such as operating at the level of the operating system, allowing the creation of multiple isolated application environments that refer to the same core. Containers do not emulate the full virtualization of hardware like hypervisor-based VMs, which allows them to be small, fast, and portable. In addition, applications developed in containers already include in these the libraries and dependencies necessary for their execution, allowing their portability to different environments without impact on their operation. In order to guarantee the isolation of computational resources between host and containers, this virtualization model uses the components namespasces and Control Groups (cgroups) of the Linux kernel (Bernstein, 2014). Figure 4 graphically represents the container virtualization model.

Figure 4 – Container virtualization and isolation characteristics.



Source: Adapted from (MICROSOFT, 2018)

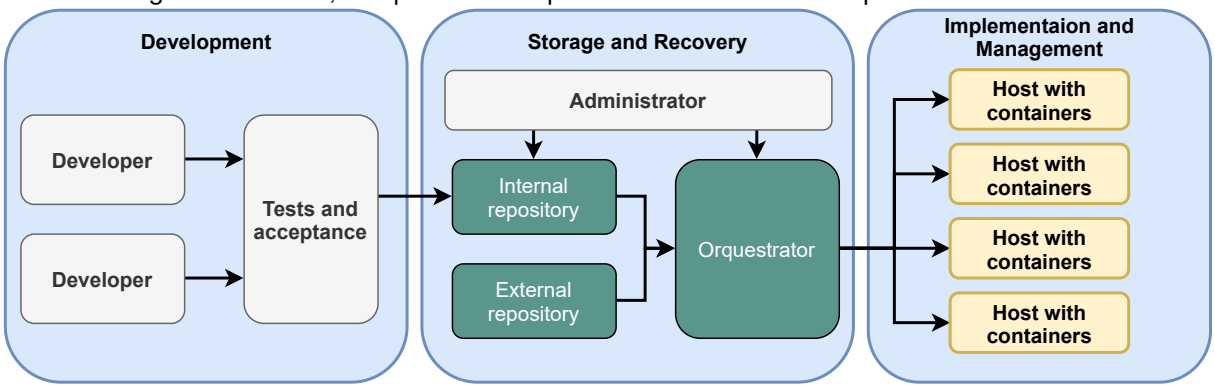
In this model, the container mechanism makes the interface between the components of the environment (*e.g.*, Clients, Orchestrator and Registry) and the OS (cgroups and namespaces). The namespasces allows to isolate processes and resources hierarchically at levels and sub-levels. cgroups controls access to resources by processes and has been part of the Linux kernel since 2007. In 2013 cgroups was redesigned, allowing the creation of container managers or orchestrators (NETTO et al., 2016), which are considered the standard for microservices in the cloud (VAUCHER, 2018). Thus, the adoption of the microservice architecture allowed to increase the performance of applications, fragmenting them into independent parts of development, versioning, provisioning, and scalability (JAMSHIDI, 2018).

These applications typically involve hundreds or even thousands of containers, making the management and monitoring of this decentralized environment a challenge, due to the high number of computational resources involved and the volume of infor-

mation generated and exchanged. This challenge led to the development of tools for container orchestration, designed to manage the deployment of containerized applications in large-scale agglomerates, being able to perform hundreds of thousands of jobs on different machines (RODRIGUEZ; BUYYA, 2018).

Another feature of the container virtualization model is that the containers are created from images, which are packages that include everything needed to be executed, such as codes, libraries, configurations, and system tools. Images are build from overlapping layers from the Base Layer, typically a minimalist distribution of an operating system (BUI, 2015). Figure 5 presents the container-based development architecture model proposed by NIST.

Figure 5 – Actors, components and phases of container development architecture.



Source: (CHANDRAMOULI et al., 2017)

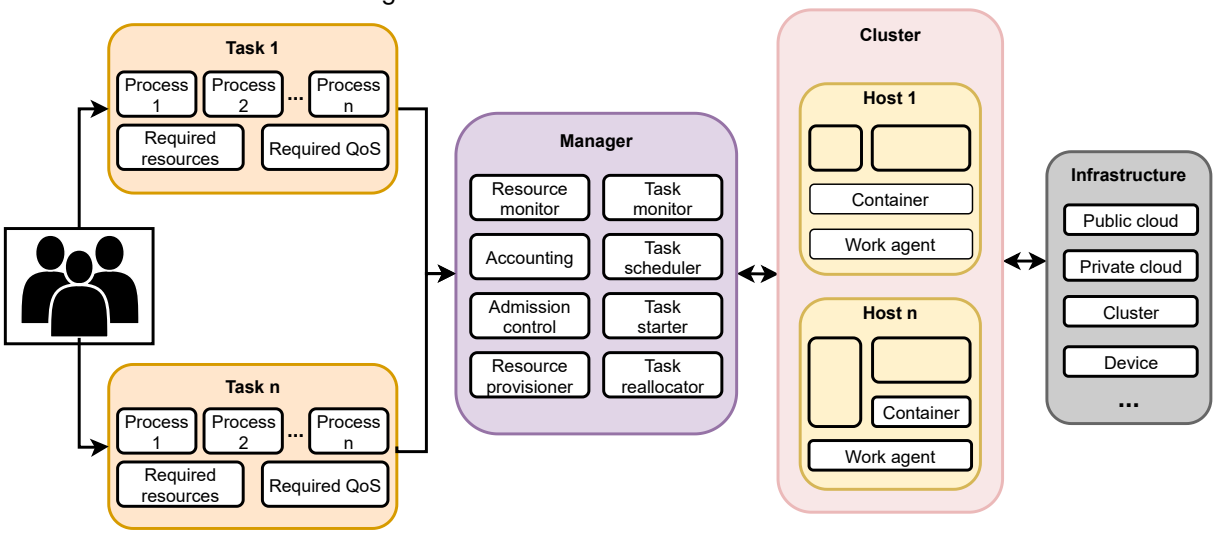
The gray components (Developers, Test and Acceptance, and Administrator), despite being outside of the container virtualization architecture, have important interactions with it. In green (Internal Repository, External Repository, and Orchestrator) are the main components of the architectural model. In yellow are the hosts in which the containers are allocated. Finally, in blue are represented the three main phases of the life cycle of the container-based development architecture.

One of the main use cases of the container virtualization model is the microservice architecture. In this model, an application is composed of small and independent services that communicate via the network and that can be implemented individually (NEWMAN, 2015). This type of architecture is opposed to the monolithic model, where a large unitary solution contains all the services. This new approach allows greater agility during the development and implementation of the application. The decomposition of an application into smaller functional parts increases its complexity by requiring the dynamic allocation of microservices. This function is performed by orchestrators, who are also responsible for the management, implementation, and scheduling of workloads of containerized applications.

2.3 ORCHESTRATORS

Most orchestration tools allow scheduling tasks for multiple users on a set of shared computing resources, optimizing their use (RODRIGUEZ; BUYYA, 2018). These tools automate the process of obtaining and implementing images, as well as managing the containers in execution (CHANDRAMOULI et al., 2017). Container orchestration defines not only its initial implementation but also the continuous management of multi containers as a single entity (PAHL et al., 2017). In addition to these features, the orchestrators also include monitoring, load balancing, and continuous workload functions, supporting the cloud infrastructure lifecycle (PALADI; MICHALAS; DANG, 2018). There are different models and technologies for container orchestration, among which Docker, Kubernetes, and LXC stand out. However, despite this variety, there is a set of features and functionality common to all. Figure 6 represents a generic model of container orchestration.

Figure 6 – Orchestrator's reference model.



Source: (RODRIGUEZ; BUYYA, 2018)

In this model, the first layer corresponds to the submission of tasks by users. Then, these tasks are sent to the Manager, that, among other functions, must provision resources and schedule the tasks to be executed on any of the nodes available in the computational cluster. Finally, this node will be responsible for the execution of the assigned task.

2.3.1 Orchestrator layers

In general, the architecture of container orchestration systems has four main entities or layers: Tasks, Manager, Computational cluster, and Infrastructure.

1. **Tasks:** Applications are sent for execution in the form of tasks. In general, they belong to several users and can range from long-term services that require low

latency to short, resource-intensive tasks. Tasks may be composed of one or more smaller processes, usually homogeneous and independent. Requirements such as processing, memory, and fault tolerance, among others, can be included in the task definition.

2. **Manager:** The Manager acts as a high-level controller, responsible for the control and management of container clusters in a group of hosts that compose the cluster. It is responsible for providing functions for organizing, implementing, managing, and securing a diverse set of containers, their dependencies, and interactions. Similar containers are grouped into sets called Coherent Units (CUs), and are managed as a single entity (SYED; FERNANDEZ, 2017). Among the components of this architectural model, the manager is primarily responsible for the system orchestration and is made up of smaller modules with different tasks. The Resource Monitor module collects and provides resource allocation and use data from all the working nodes in the computational cluster, accessed by the other components of the system. The Accounting component provides metrics relevant to cluster management, such as energy consumption, resource utilization, quantity, and type of tasks. The Admission Control module is responsible for accepting, rejecting, or prioritizing the tasks sent. For this, it must assess whether there are sufficient resources for execution and if the user's resource quota is sufficient. The Task Scheduler is responsible for the distribution of tasks in the resources of the computational cluster. When a task needs to be relocated, the module responsible is the Task Relocator, which can also discard the task, depending on the pre-defined policy. After scheduling a task, the module responsible for executing the container for this task is the Task Launcher. Finally, the Resource Provider is responsible for manually or automatically adding new nodes to the cluster.

3. **Cluster:** The Cluster is composed of worker nodes, which are nodes available to receive and perform tasks. These nodes have a component called Work Agent, responsible for actions such as collecting consumption metrics and reporting them to the Manager, starting and stopping, and monitoring tasks deployed at the node.

4. **Infrastructure:** The Infrastructure is composed of a set of hardware and software that guarantees the execution of the environment. This virtualization model offers flexibility in terms of infrastructure, which can be public or private, in addition to having both physical and virtual machines in their composition.

2.4 EVENT TRACKING IN CONTAINER OPERATIONS

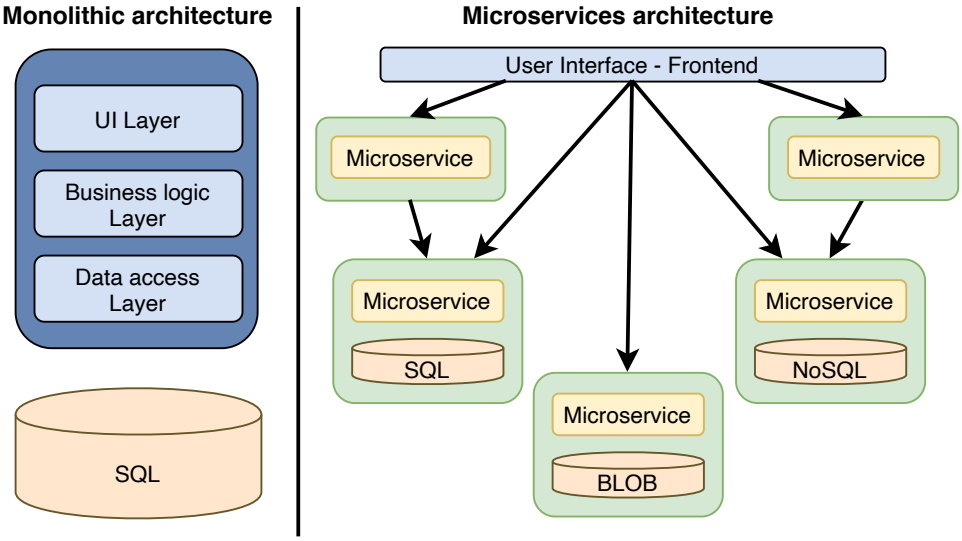
With the emergence of cloud computing providers (*e.g.*, Microsoft Azure and Amazon Web Services), companies started offering computing resources, platforms, and applications on-demand, allowing developers and partners to offers and sell their software and products to clients, running them on the cloud computing platforms made available by the providers. Cloud computing environments that uses a container virtualization model allows the implementation of solutions and applications based on microservices, the basic element resulting from the architectural decomposition of an application into loosely coupled standards. The microservices consists of independent services that communicates using a well-defined standard communication protocol and set of Application Programming Interfaces (APIs), regardless of vendor, product or technology (KARMEL; CHANDRAMOULI; IORGA, 2016). In this model, despite the independence of services, there is still a minimum centralization, responsible for their management, which can be developed in different programming languages and use different data storage technologies.

The microservices architecture was developed as a counterpoint to the model known as monolithic, where an application is developed as a unit, composed of three parts: client-side user interface, database, and a server-side application. This server-side application is called "monolithic", that is a single executable that aggregates all the functions of the application. In this way, any change in the system involves the development and implementation of every application on the server-side. With the popularization of cloud computing and the adoption of the containerized virtualization model, this model showed deficiencies due to the difficulty in maintaining a good modular structure. In addition, scaling a monolithic application requires scaling the entire application, consuming excessive and often unnecessary resources (O'CONNOR; ELGER; CLARKE, 2017).

These characteristics led to the microservice architecture model, in which the independence of the services that make up the applications allows flexibility both in its development and in its implementation. In addition, the possibility of highly granular modularization allows scaling and defining different levels of redundancy for each service, in addition to enabling the development of services in different programming languages. Figure 7 presents a comparison between the architecture models described.

In the microservice architecture model, it is possible to visualize the independence of the containers. In this model, each application is composed of several individualized services in containers, triggered on demand through the user interface. This is possible because each container has all the dependencies and libraries necessary for the execution of the service it has. These characteristics have led to a considerable

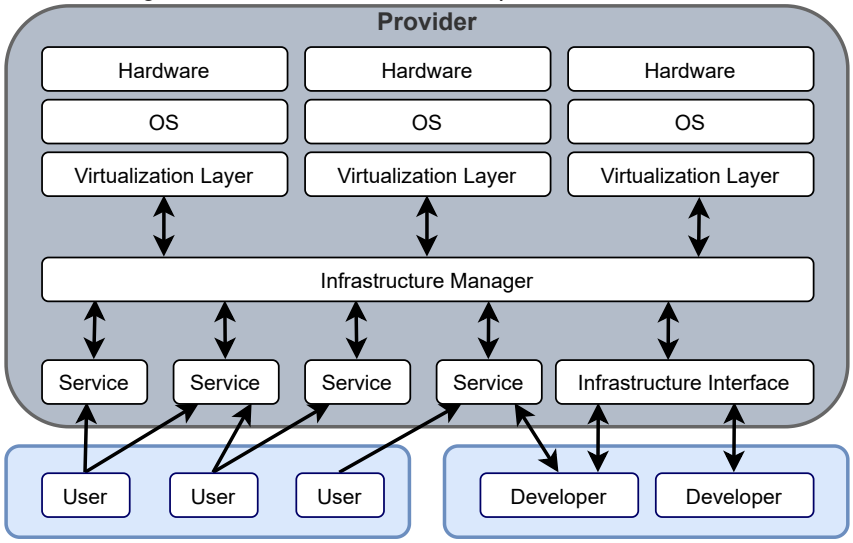
Figure 7 – Comparison between models of Monolithic architecture and Microservices.



Source: Adapted from (WU, 2017)

increase in the adoption and popularity of the microservice architecture. Containers are now widely used by organizations to implement a growing and diverse workload of applications such as web, Big Data, and Internet of Things (IoT) services in public or private data centers. This, in turn, led to the emergence of container orchestration platforms designed to manage the deployment of containerized applications in large-scale clusters, being able to perform hundreds of thousands of jobs on different machines (RODRIGUEZ; BUYYA, 2018). Such characteristics result in great complexity for the management and monitoring of this environment due to the high number of computational resources that compose it and services in execution, generating, and passing information. Figure 8 represents the scenario of interactions between the client, provider, and proposed developer.

Figure 8 – Interactions between provider and clients.



Source: (VAQUERO et al., 2009)

In this model, there are interactions between three actors: the application developer, the cloud computing platform provider, and the application client. Compared to the reference model in cloud computing, proposed by (BOHN et al., 2011), the consumer role is played by both the developer and the client of the application. However, the reference model foresees that the roles of auditor and broker, responsible for auditing the use, performance, and security of services (BOHN et al., 2011), are performed independently and reliably. According to the model proposed by (BARDSIRI; HASHEMI, 2014), the metrics evaluated in a cloud virtualization environment are distributed in three main aspects: performance, financial, and security. In this way, data referring to several metrics must be collected and stored in a safe way to allow the subsequent analysis of the environment and services in execution, being important the existence of a mechanism that promotes consensus between the parties regarding the collected information.

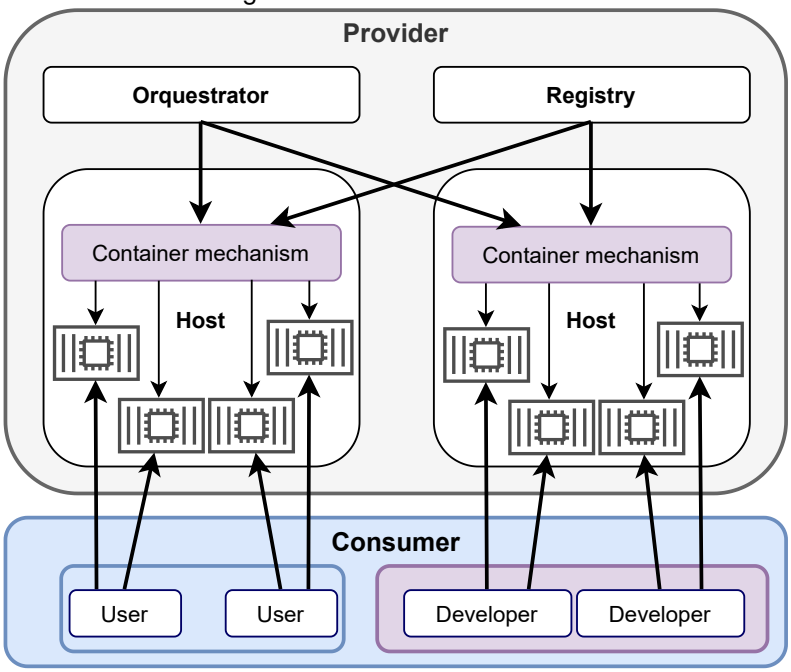
2.5 PROBLEM DEFINITION

In a cloud computing scenario, in which services are developed and offered to customers on an outsourced platform, the monitoring and management of this environment is an essential aspect of the infrastructure that makes it possible to improve the scalability and distribution of services, detection, and prevention of failures, performance analysis, among others (JIMÉNEZ et al., 2015). In this context, the act of monitoring consists of observing the execution of the virtual environment, collecting and making available for analysis, periodically, a set of predefined variables.

In general, the cloud computing provider offers its monitoring system to the customers. On the other hand, customers can implement an independent monitoring service (DAWADI; SHAKYA; PAUDYAL, 2017). This possibility brings the autonomy of the actors concerning monitoring as a benefit, being able to adapt it to their needs. However, this autonomy also brings the possibility of disagreements between the actors regarding the information collected, which may lead to different analyzes of the same scenario. Specifically considering container lifecycle events, divergences in the collection of the startup, suspension, and termination events can result in inaccurate analyzes of the environment and applications being executed.

In addition, centralized monitoring solutions directly depend on trusting that actor to guarantee the integrity of the collected data. Thus, one of the actors can change the data collected by their monitoring, putting the other results in doubt. Figure 9 represents the proposed scenario, in which a distributed application, hosted on a cloud computing provider, is offered by a developer to a customer.

Figure 9 – Use case scenario.



Source: Author.

In this scenario, there is an interaction between two actors: Provider and Consumer. However, the consumer is made up of both the developers, who host their applications at the cloud computing provider and the users of the applications. Each of these actors has different perspectives and needs:

- **Cloud computing provider:** For the cloud computing provider, monitoring is a key tool for managing hardware and software infrastructures. Specifically dealing with the collection of container life cycle events, it is through it that the provider can chronologically identify the containers in execution, as well as their interruptions and returns.

  This data allows the provider to determine the resources used over a period, which is especially important in billing models based on resource consumption. In addition, collecting interruptions and returns in the execution of containers is useful in failure analysis and audits.

- **Consumer: Developer:** For the developer, monitoring the life cycle of their applications is important from the point of view of development, optimization, and billing. The collection and analysis of the life cycle events of the containers allow to track and identify failures in the services of the applications. Also through these data, the developer can check the execution time of the services, facilitating the optimization of the application code. In addition, in models in which charging is linked to the consumption of computational resources, the developer needs to monitor the applications to calculate their respective execution costs.

- **Consumer: User:** From the application user's point of view, monitoring life cycle events is important to identify failures and outages, scalability, and pricing. The monitoring of service failures by users allows agility in decision-making in case of any problems. As for performance, the analysis of the event logs can validate whether the number of instances running is sufficient for the demand. Finally, from the point of view of charging, the collection of container life cycle events is important for the user to evaluate the number of instances and the respective execution time of the application services that he uses, being able to validate whether the charge generated by the service provider cloud computing services is by the monitoring record (WARD; BARKER, 2014).

The possibility for each actor to have its monitoring mechanism gives rise to possible disagreements regarding the events collected. In these situations, the question arises as to which of the different collected data is correct. Thus, the problem presented by this work is how to guarantee the validation and consensus among the actors participating in container virtualization environments about container and services life cycle events order and integrity so that it is possible to audit them by any of the parties, and how to maintain this data to preserve its integrity, authenticity, availability, and irreversibility.

2.6 CHAPTER CONSIDERATIONS

The cloud computing model allows computing resources to be shared on-demand using the network. Virtualization, by its side, allows the abstraction of hardware, offering new development models with the use of modules and packages, providing new services quickly and with load balancing and scalability. Container-based virtualization quickly gained prominence, making it the top choice for new application developments in cloud computing environments. Among its main characteristics, agility and portability stand out.

Orchestrators are tools that allow the scheduling, implementation, and continuous management of tasks by multiple users on a set of shared computational resources. Through it is possible to manage a cluster of hosts that executes the container mechanism. Among its functions are maintaining the state of the cluster, scheduling services, and providing an API. Through the API it is possible to collect data related to the container life cycle events in the container virtualization environment.

Monitoring a cloud virtualization environment is a complex activity, depending on the periodic collection and storage of data about the environment. Issues such as performance analysis, efficiency in resource allocation, and pricing based on consumption of computational resources directly depend on the reliability of the collected data.

These different perspectives and needs have in common the requirement for complete, available, and auditable data, in addition to the consensus among the actors on the validity of the stored data. Literature reviews on cloud computing and container monitoring highlight as main research questions those linked to the scalability, elasticity, architecture, and precision of monitoring solutions. However, few studies have as a research question the reliability of the data collected by the monitoring agents. The motivation to carry out the monitoring varies according to the perspective (*i.e.*, Provider, Developer, or User), but it depends directly on the guarantee of confidentiality, integrity, and availability of the collected data.

3 REQUIREMENTS AND PROPOSAL

Base on the definition of the problem stated, it is possible to specify the functional and non-functional requirements to be met by the proposed solution. These requirements address not only the solution of the container lifecycle monitoring problem but also the security of the collected data. With the requirements definition is possible to search for related works as well as identify the existence of similar tools, evaluating the adopted approach. This analysis allows evaluating, among the identified related works, the fulfillment of the specified requirements.

Section 2.5 presents the two main types of monitoring solutions for containerized virtualization environments: centralized in the Provider or implemented individually by each of the actors. While centralized monitoring requires unrestricted trust in one of the actors, prevents integrity checking in the event of a failure, and limits customization capabilities, deploying several different monitoring tools can result in conflicts between the collected data.

In this sense, the analysis of the problem gains relevance for proposing an alternative or complementary monitoring solution to the existing models. This solution seeks to unite the main benefits of the two original solutions (*e.g.*, single database and possibility of customizing the solution according to the actors' needs), adding, however, mechanisms that mitigate the main vulnerabilities existing in monitoring solutions, discussed in Section 3.4. In general, attacks on these solutions seek to affect the reliability, integrity, and authenticity of records, either by delaying or preventing their collection or even through tampering or generating false records. Thus, the proposed solution must have resources that guarantee the safety of data collection, communication, and storage processes.

3.1 REQUIREMENTS SPECIFICATION

The problem presented occurs in a virtualization environment based on containers and implemented in a computational cloud, which provides a greater degree of dynamism than common virtual machines. Therefore, it is necessary to reproduce this environment so that it is possible to implement and evaluate the proposed solution. Thus, for the development and implementation of the proposed solution, we consider the following scenario:

- A virtualization environment in containers in a computational cloud, with a containerization platform (*e.g.*, Docker) and use of an orchestration tool (*e.g.*, Docker

Swarm);

- A set of nodes, each one playing the role of Provider, Developer, and User, thus ensuring the decentralization of the processes that make up the solution;
- Access to a data storage repository to store the collected lifecycle events; and
- Access to the orchestrator API with permission to execute, update, suspend and terminate services and containers, generating events to be collected by the proposed solution.

By meeting the prerequisites established in the proposed scenario the necessary conditions for the development and implementation stages are guaranteed and it is possible to specify the set of functional and non-functional requirements that make up the proposed solution.

3.1.1 Functional Requirements

The functional requirements (RF), defined based on the criteria presented in Section 2.5 are:

- **FR1:** Perform the collection of all “container” and “service” type events in the cluster nodes;
  - The solution must collect all occurrences of “service” and “container” events referring, respectively, to the implemented services and the containers running in the nodes that make up the computational cluster. Specifically, for type “container” the events to be collected are: *create*, *start*, *pause*, *unpause*, *kill*, *terminate* and *destroy*; and for the "service" type they are: *create*, *remove* and *update*;
- **FR2:** Ensure the integrity of the collected data;
  - The solution must implement mechanisms that ensure data integrity throughout the process so that it is possible to identify data manipulation;
- **FR3:** Ensure the authenticity of the processes and components;
  - The solution must implement authentication mechanisms for the collection, ordering, and validation processes and components;
- **FR4:** Ensure the security of communication between the solution processes;
  - All communication carried out during the collection, validation, and storage processes must use secure protocols to mitigate the risk of data leakage;

- **FR5:** Chronologically order the records, ensuring consensus among the actors; and
  - The solution must implement mechanisms that allow reaching consensus among the actors regarding the chronological order of the collected events.
- **FR6:** Store the event records in a distributed repository, protected from tampering and accessible to all actors, facilitating data auditing.
  - The solution must store the validated records in a distributed repository and that uses mechanisms that ensure data integrity so that it is possible to audit and track any attempt to manipulate the data.

3.1.2 Non-functional requirements

Non-functional requirements (NFR) identified:

- **NFR1:** Adjust the volume available for storing the events collected according to the size of the environment and the number of applications and containers running; and
- **NFR2:** The solution must operate in a non-intrusive manner, that is, the general performance of the applications must not be affected by its operation.

From the defined functional and non-functional requirements, it is possible to assess how related works address the proposed problem.

3.2 RELATED WORK

Monitoring container virtualization environments is a topic of significant importance from a technological, administrative, and business point of view. Therefore, this is a recurrent subject in several academic works and also exists commercial and open source solutions that deal with this subject. For the present work, the research method adopted to identify related work is systematic mapping, which aims to obtain an overview of the research area, identify and quantify evidence (KEELE et al., 2007). With the results obtained by the mapping, the aim is to identify the works related to the research area in which the proposed problem fits, analyzing whether they meet the defined functional requirements.

3.2.1 Related Work Selection

The search for related works started with the definition of a set of keywords to be applied in scientific search engines was defined. The keywords definition intended to

highlight the focus on monitoring virtualization environments in containers that use the Docker platform. Although there are other virtualization platforms in containers, Docker is the reference and, therefore, it is likely to be mentioned by other works dealing with the same theme, improving the overall assertiveness of the search. In order to format the search expression the logical operators *AND* and *OR* were used, in addition to the elaboration of the phrasing using parentheses. Thus, the keywords used were *docker*, *container* and *monitoring*, forming the search expression: *(docker OR container) AND monitoring*.

The search engines used to carry out this research were: Google Scholar, ACM Digital Library, IEEE Xplore, and Springer Link. Google Scholar, for conducting a broader search, was the engine that presented the highest amount of results. The term container, present in the search expression, presented ambiguity in some search engines, resulting in works with a theme related to physical cargo transport containers (*e.g.*, *container shipping*). As mentioned earlier, the use of the term Docker reduced the number of ambiguities. In addition, for the Google Scholar engine, we aim to refine the search by excluding works with the term "shipping" using the expression: *monitoring (docker OR container) and virtualization -shipping*.

3.2.2 Inclusion and Exclusion Criteria

Among the works resulting from the search, a set of inclusion and exclusion criteria were applied to delimit the results obtained. Criteria are summarized in Table 1.

Table 1 – Inclusion and Exclusion Criteria.

Inclusion Criteria	Exclusion Criteria
Works written in English or Portuguese	”Gray” literature
Articles, abstracts, book chapters and technical reports	Publication prior to 2012
Monitoring in a virtualization environment	

Source: Author.

Firstly, works written in English or Portuguese were defined as inclusion criteria. The second inclusion criterion establishes that the selected works must be articles, extended abstracts, book chapters, or technical reports. Finally, the papers should address the topic related to virtualization environment monitoring, allowing for an assessment of how the monitoring topic is applied in other virtualized environments besides containers. In the case of the identification of duplicated works, only the most recent result should be considered.

Exclusion criteria were ”gray” literature works (*i.e.*, publications in blogs and journals without scientific rigor), and productions in a language that is not among the pre-defined ones. Another exclusion criterion defined was concerning the publication

date. As the present work presents a proposal aimed at the container virtualization environment, only the results with a publication date since 2012, the year in which the Docker technology was developed, reference of the model, were considered.

3.2.3 Search Results

The search was carried out in four pre-defined academic search engines, using the elaborate search expression, resulting in an initial total of 57318 results. Filters referring to the publication date and other inclusion and exclusion criteria were applied, reducing the result to 20001 works. The analysis of the results found that many of these works still did not correlate with the proposed theme. Thus, a sample was selected with the first 100 results from each search engine, ranked by relevance. Then, the titles and keywords of these works were analyzed, resulting in a set of works for reading the abstract. After reading the abstracts, ten papers related to the proposed theme were selected to a full read, among which five were identified as related work. In addition to the identified works, two open-source tools were also considered in this analysis: Collectd and cAdvisor. Thus, the following list presents the related works and tools identified:

- **1 - Light-Weight Service Lifecycle Management For Edge Devices In IIoT Domain** (Jo; Ha; Jeong, 2018): The proposed work presents a solution for the management of the container life cycle using Docker.
- **2 - CoMMoN: The Real-Time Container and Migration Monitoring as a Service in the Cloud** (DAWADI; SHAKYA; PAUDYAL, 2017): This solution brings a tool for collecting and storing metrics and events, as well as migration of Docker containers.
- **3 - Automated deployment of a microservice-based monitoring infrastructure** (CIUFFOLETTI, 2015): This proposal describes a microservices monitoring framework "*as a service*".
- **4 - A Cloud-native Monitoring and Analytics Framework** (OLIVEIRA et al., 2017): The proposal presented details a framework for collecting and storing container metrics in a distributed repository.
- **5 - Logchain: Blockchain-assisted Log Storage** (POURMAJIDI; MIRANSKYY, 2018): The described solution uses a blockchain-based logging system, in order to ensure data immutability.
- **6 - CollectD** (COLLECTD, 2020): A daemon that collects periodic information and provides mechanisms to store and monitor the collected data.

- **7 - cAdvisor** (CADVISOR, 2020): A daemon that collects, gathers, processes, and exports information about running containers.

3.2.4 Related Work Analysis

The related works prepared by (Jo; Ha; Jeong, 2018; DAWADI; SHAKYA; PAUDYAL, 2017; CIUFFOLETTI, 2015; OLIVEIRA et al., 2017) present different proposals for the monitoring of containers in a Docker environment. However, none of the four jobs meet FR3 and FR4. The proposal presented in (POURMAJIDI; MIRANSKYY, 2018) is not a specific model for monitoring containers, but computational clouds. In addition to them, two open source solutions available and used for the same purpose were identified and related in the analysis: cAdvisor and Collectd. Table 2 presents a comparison between the functional requirements of the proposed solution and the related works identified.

Table 2 – Related works: Functional Requirements attendance.

	1	2	3	4	5	6	7
FR1 - Event Collection	Yes	Yes	Yes	Yes	No	Yes	Yes
FR2 - Data integrity	No	No	No	No	Yes	No	No
FR3 - Authenticity	No	No	No	No	No	No	No
FR4 - Communication security	No	No	No	No	No	Yes	Yes
FR5 - Chronological order	No	Yes	No	No	Yes	Yes	Yes
FR6 - Distributed storage secure and auditable	No	No	No	No	Yes	Yes	Yes

Source: Author.

The FR1 is served by all jobs and tools, except (POURMAJIDI; MIRANSKYY, 2018), which does not focus on collecting container events. As for FR2, the only work that deals with data integrity is (POURMAJIDI; MIRANSKYY, 2018), through the cryptographic chain adopted in the proposed blockchain. About FR3, (Jo; Ha; Jeong, 2018) states that the client component (Pharos Node), when installed, performs a registration process on the server (Pharos Anchor). However, the process is not described and it is not possible to state that some authenticity implemented mechanism. In (POURMAJIDI; MIRANSKYY, 2018), the solution implements the concept of blockchain as a log repository. The work does not mention, however, the adopted mechanisms to guarantee the collection and storage processes authenticity. Open source tools, in turn, allow the integration of authenticity mechanisms but do not natively bring such an option. The other works identified do not describe mechanisms to guarantee the authenticity of the processes carried out.

As for FR4, the only one of the related works identified that addresses the topic is (Jo; Ha; Jeong, 2018), which proposes the use of *webhooks* (user-defined HTTP callbacks). However, the work does not use, in communication, a secure protocol (*e.g.*,

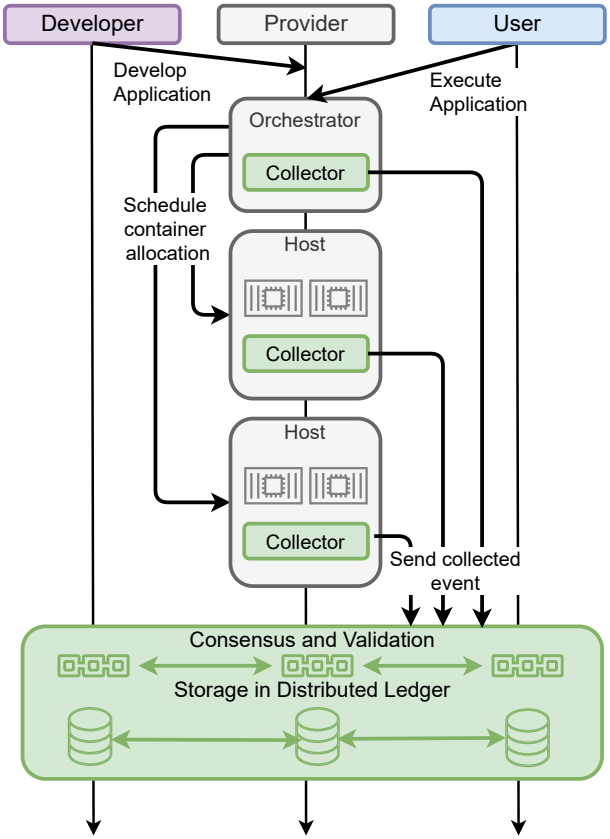
Transport Layer Security (TLS)), making data interception possible. The (COLLECTD, 2020) and (CADVISOR, 2020) tools allow integration using secure protocols. FR5, in turn, is served in (DAWADI; SHAKYA; PAUDYAL, 2017) through the use of an Time Series Database (TSDB). In (CIUFFOLETTI, 2015) the chronological order is considered, but there are no details about the applied process. In (OLIVEIRA et al., 2017) the collectors (*i.e.*, *crawlers*) add a timestamp to the collected data. This mechanism, however, is subject to synchrony problems between the collectors' clocks. In (POURMAJIDI; MIRANSKYY, 2018), during the blocks generation, they receive a *timestamp* that composes the *hash* of the block, ensuring validation and consequently consensus among the actors. Finally, FR6 is served by (POURMAJIDI; MIRANSKYY, 2018), which uses storage in *distributed ledger*. The (CADVISOR, 2020) and (COLLECTD, 2020) tools export natively to various repository models, including distributed.

In this way, with the analysis of related works, it is possible to assess that the research problem is not fully addressed by any of the identified solutions. The solution presented in (POURMAJIDI; MIRANSKYY, 2018) is the one that comes closest to fully meet the functional requirements. However, the proposal presented by this work is generic, without going into details about the type of permission and the consensus mechanism used. It is also modeled ”as-a-service”, and the data collection process is outside the scope of the solution. The open-source tools cAdvisor and Collectd, in turn, have as a differential the storage capacity in distributed repositories, but they do not have mechanisms to guarantee data integrity and chronological consensus.

3.3 PROPOSED SOLUTION

As described in Section 2.5, the monitoring of distributed applications running in containerized virtualization environments can be performed by the Provider or individually, by any of the actors involved. In the centralized model, participants depend on the security of processes carried out by the Provider. On the other hand, with several different monitoring solutions in the same environment, the possibility of divergences between them increases, either due to failures or attacks. Thus, this work presents an alternative solution, called event2ledger, which allows the joint and consensual participation of actors in the processes of validation, ordering, and storage of events. Figure 10 represents the flow of generation of container and service lifecycle events, as well as the main actors, relationships and actions involved, indicating how the proposed solution interacts with the elements of the environment.

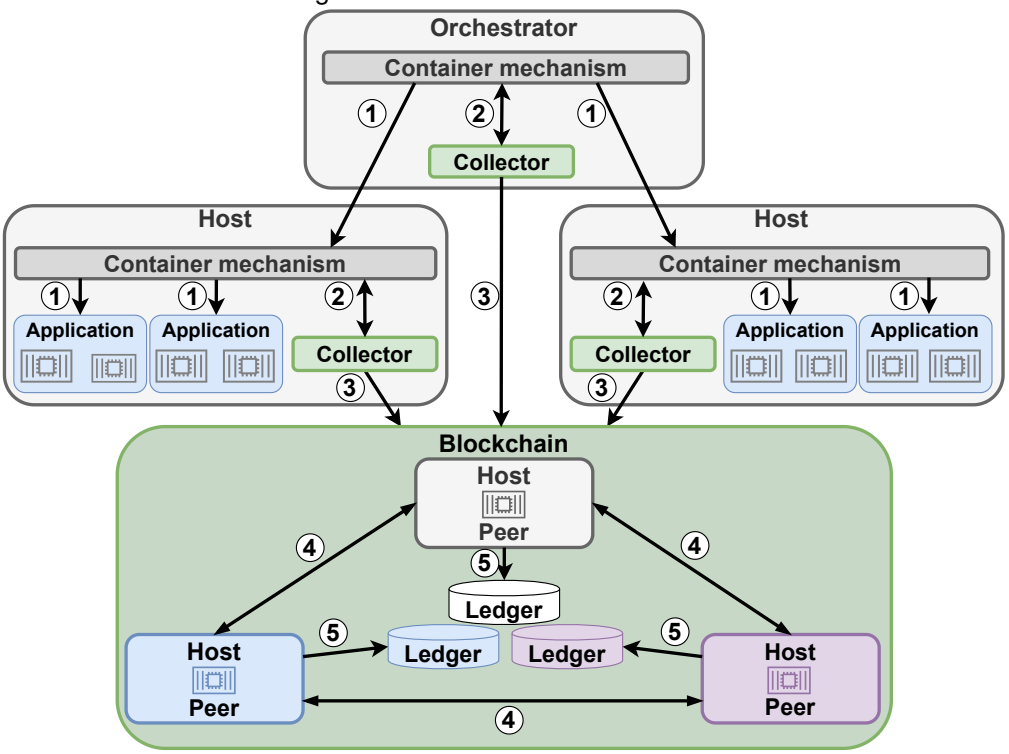
Figure 10 – Proposed solution's sequence diagram.



Source: Author.

As some orchestration mechanisms implement different types of hosts (*e.g.*, for cluster management and container execution), collectors must be installed in both to obtain both application events (”service type ”) and containers (type ”container”). Thus, the proposed proof of concept scenario is composed of three actors (Provider, Developer, and User), with the Provider being responsible for the execution and maintenance of the three hosts that compose the cluster, the container mechanism, orchestrator, and other infrastructure that make up the environment. The Developer, on the other hand, is responsible for creating applications that are available for User execution on the cluster hosts. Each cluster host executes an instance of the even2ledger collector that sends the collected events from the local host to validation and storage. Finally, they are stored in a distributed repository, of which each actor has an instance containing the complete copy of the data and which implements integrity and availability guarantee mechanisms. The Figure 11 represents, in a generic way, the proposal presented by the present work.

Figure 11 – Generic solution model.



Source: Author.

The generic model of the solution considers a cluster composed of three hosts, one of them being responsible for the orchestration and the others for the execution of the containers that compose the applications and that generate the events (1). The event2ledger, in turn, has a set of collectors that connect directly to the container mechanisms running on the hosts and receive the generated events in real-time (2). This direct connection with the container mechanism allows the non-intrusive event collection, thus meeting FR1 and NFR2. To guarantee the collector authenticity and comply with FR3 the solution proposes the use of asymmetric encryption. For this, the collector signs the collected event with its private key, which will be validated in the future by the actors that compose the proposed scenario. After generated and signed, the transaction is sent to the nodes that make up the blockchain (3). Each solution component has its certificate, used to establish a secure TLS connection during all the communication processes, as demanded by FR4. The collected and validated events are then forwarded to a consensus mechanism that allows the actors to agree on their order (4) in compliance with FR5. Finally, the events are stored in a distributed repository (5) composed of three instances belonging to the actors (*i.e.*, Provider, Developer, and User). The fact that each actor has a complete copy of the repository, together with the cryptographic chaining mechanism and the permissions policies, allows the auditing and tracking of changes to the stored data, in compliance with FR6. Table 3 presents the approach adopted by the proposed solution to meet the established functional requirements.

Table 3 – Functional Requirements compliance.

Functional Requirement	event2ledger
FR1 - Event Collection	Complies using container mechanism
FR2 - Data Integrity	Complies using cryptographic chaining and TLS
FR3 - Authenticity	Complies using asymmetric cryptography
FR4 - Communication security	Complies using TLS
FR5 - Chronological Order	Complies using consensus mechanism
FR6 - Distributed storage secure and auditable	Complies using blockchain

Source: Author.

In addition to meeting the functional requirements, event2ledger also meets the defined non-functional requirements. The event2ledger implementation foresees compliance with NFR1 through the adequate sizing of the available volume for the nodes responsible for storing the events. To be compliant with NFR2 the solution interacts with the container mechanism to receive the flow of generated events in real-time and non-intrusively.

3.4 ATTACK MODEL

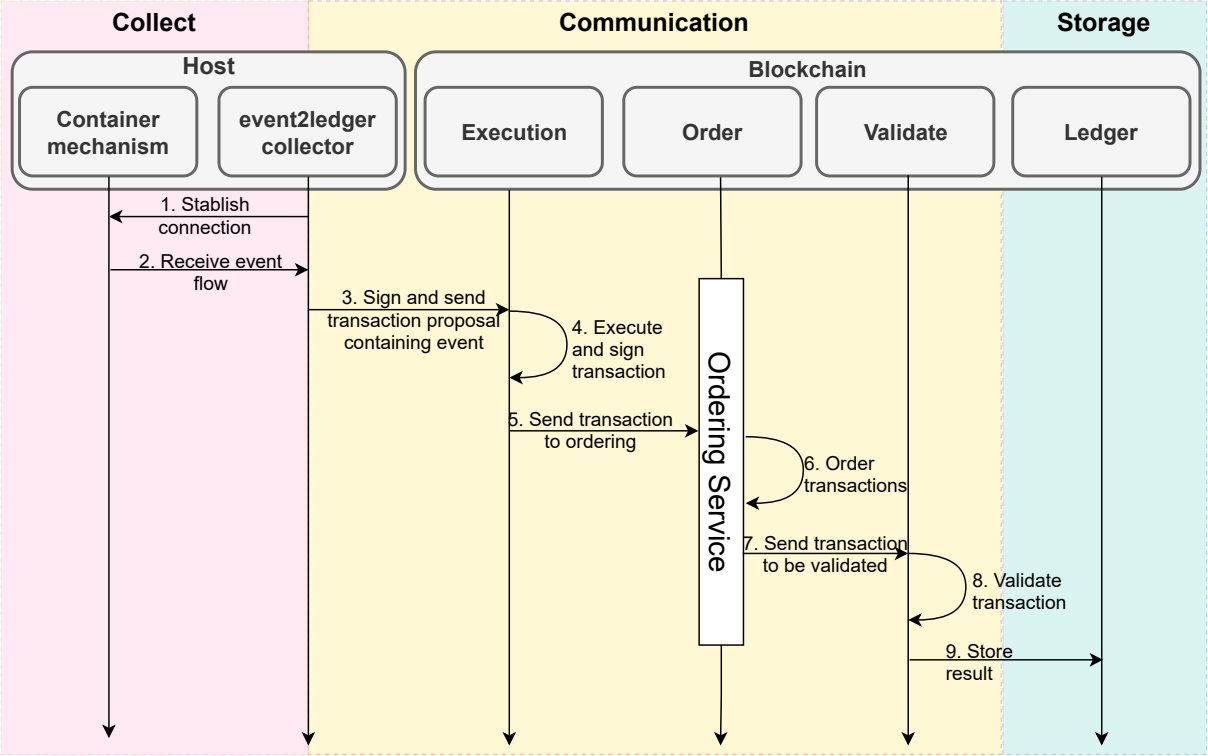
Vulnerabilities are weaknesses in a system, system security procedure, internal controls, or implementation that can be exploited by a source of threat (NIELES; DEMPSEY; PILLITTERI, 2017). Vulnerabilities leave systems susceptible to many activities that result in significant and sometimes irreversible losses for an individual, group, or organization. Attacks, in turn, are actions taken by attackers who use tools or techniques to exploit vulnerabilities to damage a system or interfere with its operations.

As presented in Section 2.5, the architecture of monitoring solutions commonly used in containerized virtualization environments can be centralized at the provider or implemented individually by the participating actors. But regardless of the model, monitoring solutions are based on a reliable chronological record of events, correctly representing the alternating life cycle states of applications and containers. Furthermore, it is essential to guarantee the integrity and authenticity of the collected events to carry out a precise analysis of the virtualization environment and the running applications. However, these solutions are susceptible to vulnerabilities that can impact data reliability, availability, and integrity, in addition to making it difficult or impossible to audit.

Thus, in the proposed scenario, attacks on monitoring solutions can be executed by any actor. In this sense, (RYAN, 2014) states that most cloud applications demand unrestricted trust in the Provider from consumers, a scenario considered by the author as overly optimistic. On the other hand, he claims that considering the Provider an unreliable actor is also unrealistic, after all, they are large organizations with reputations to be preserved and that compete to attract customers. So they are somewhere in between these extremes, and they are in a position that can launch an attack as long

as they leave no evidence (HENZE et al., 2017). This approach does not imply that the Provider will necessarily act maliciously but recognizes that this possibility exists, not only from the Provider but also from any actors or their employees. Thus, the approach proposed by this work presents an alternative solution, which implements the monitoring solution additional security, ordering, and consensus mechanisms performed by the actors involved. Figure 12 represents the execution flow of the proposed solution.

Figure 12 – Event2ledger solution flow.



Source: Author.

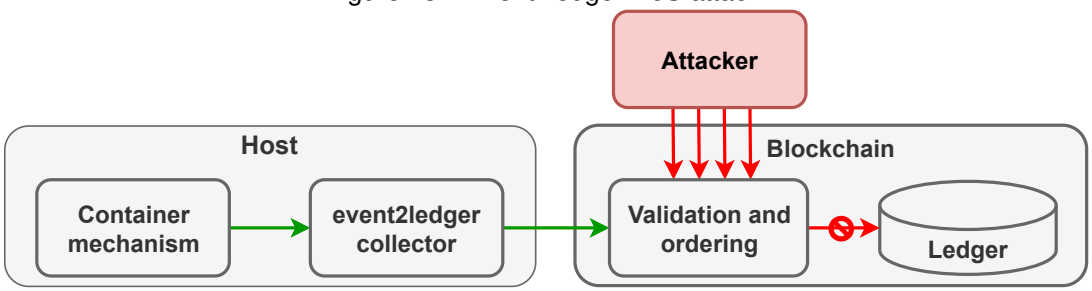
The proposed model aims to implement security mechanisms by integrating a blockchain into the solution to guarantee the integrity and reliability of the data after collection. With this modification, the monitoring process comprising the collection, communication, and storage of events becomes more complex but more secure as it contemplates authentication, ordering, and validation mechanisms carried out collectively by the actors participating in the blockchain. Thus, the elaboration of the attack model and the analysis of the implemented defense mechanisms considered the classification proposed by (HENZE et al., 2017), which brings together the attacks on monitoring solutions into four groups: Retention, Modification, Insertion, and Reordering. It is important to note that the attack model covers only the solution and its components, being out-of-scope attacks that try to indirectly compromise the solution (e.g., Denial of Service (DoS) in a switch or router can lead to solution unavailability).

3.4.1 Retention Attacks

Included in this group are attacks that delay sending collected events to prevent them from being delivered within the required time window. These attacks focus on event collection and transaction generation processes. Considering that the process of collecting events from the environment is performed in real-time, there should be a collection time window, to identify possible delays and prevent the storage of events outside the given window.

A retention attack can stem from a DoS type attack performed by a malicious agent. DoS attack is characterized by an explicit attempt to prevent legitimate users of a given service from using the desired resources, either by overloading the network, interfering with connections between machines, or preventing access by a specific actor or service (LAU et al., 2000). Considering that the event collection process performed by event2ledger is local, and realized directly from the container mechanism, the DoS attacks to the solution target the blockchain, as shown in Figure 13.

Figure 13 – Event2ledger DoS attack.



Source: Author.

The purpose of this attack is to overload the blockchain with invalid transactions to delay or even prevent legitimate transactions sent by collectors from being processed. In addition to being a result of DoS attacks, retention can also be caused by scalability issues of solution components which, if undersized, can cause bottlenecks during times of peak demand.

Thus, it is important to analyze the performance of the proposed solution to verify if it is compatible with the volume of events generated in highly dynamic environments, such as virtualization in containers. This analysis makes it possible to assess the solution's vulnerability to DoS attacks as identify any limitation of its architecture or components. Through it, it is also possible to evaluate, separately, the performance of the collector and blockchain, from the point of view of the number of transactions per second that they are capable of generating and processing.

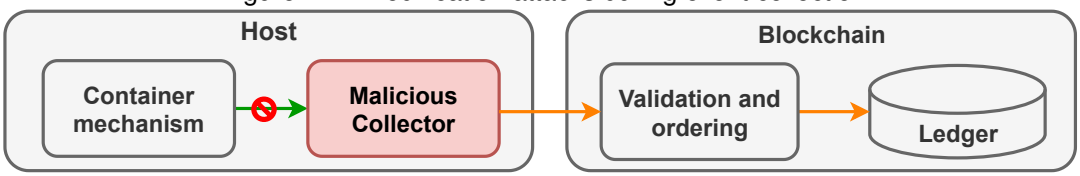
Thus, the solution must implement mechanisms that reduce the possibility of the occurrence of DoS attacks. In addition, performing performance tests after implementation aims to assess the vulnerability of the solution to typical retention attacks by identifying the (1) volume of transactions per second that the proposed collector is

capable of generating, (2) volume of transactions per second that the blockchain can process and (3) the number of collectors in parallel that the implemented blockchain supports without retention occurrence.

3.4.2 Modification Attacks

This group includes attacks that seek to alter event records to compromise the reliability of the solution. These attacks can occur in two moments: during the collection of events or after their storage. Modification attacks that occur during collection are depicted in Figure 14 and are generally due to compromised collector's private key or malicious code injection, a type of attack aimed at exploiting vulnerabilities via implementation code and gain unauthorized access, manipulate, ignore or delete data (CHOO, 2011).

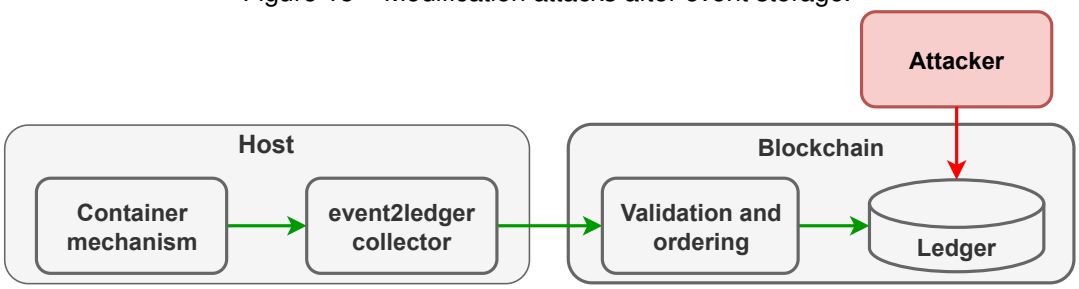
Figure 14 – Modification attacks during event collection.



Source: Author.

Modification attacks that occur after data storage exploit vulnerabilities in the repository, whether by leaking credentials, application failures, or configuration problems, to gain privileged access to manipulate the stored data, and are represented in Figure 15.

Figure 15 – Modification attacks after event storage.



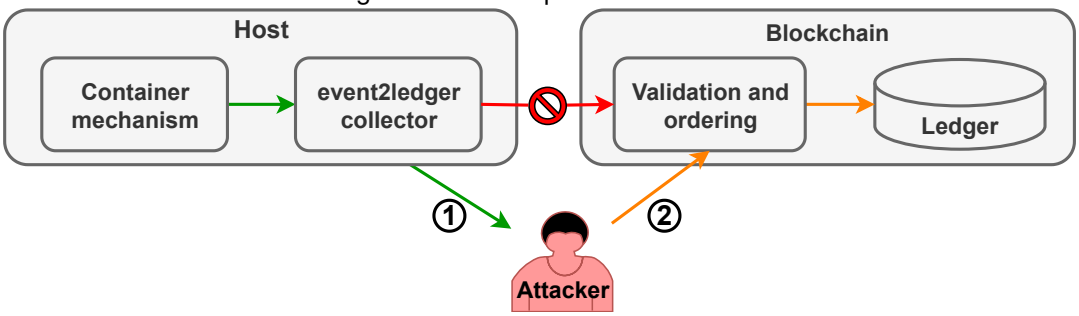
Source: Author.

In this sense, to develop a secure monitoring solution, as proposed by the RF of this work, it is essential to implement mechanisms that mitigate the risk of Modification attacks. For that, these mechanisms must guarantee that only a certain set of collectors has permission to send events to storage and to detect collector code modification. In addition, the solution must also guarantee the integrity of the stored data and allow the identification of manipulation attempts.

3.4.3 Insertion Attacks

The attacks of this group aim to create false records or even duplicating them to generate inconsistencies in the database. They can occur during the event collection step, similarly to the Modification attacks, or during the sending of collected events, through Man-in-the-Middle (MitM) type attacks, represented in Figure 16.

Figure 16 – Example of MitM attack.



Source: Author.

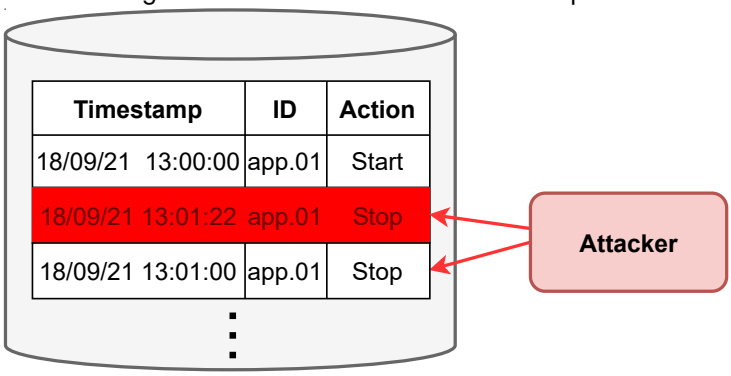
As described by (CONTI; DRAGONI; LESYK, 2016), this scenario is composed of two *endpoints*, represented by a Sender and a Receiver, and an attacker. The attacker secretly takes control of the communication channel, intercepting the received traffic (1) and modifying or inserting events before forwarding it to the Receiver (2). Furthermore, victims are unaware of the attack, believing the communication channel is secure. In this way, this type of attack is potentially dangerous because, in addition to compromising data integrity and being difficult to detect, it allows the launch of other types of attacks (*e.g.*, modification attacks). Considering that the proposed solution is based on intense communication between its components, it is necessary to implement mechanisms to grant the authenticity, integrity, and confidentiality of the messages exchanged.

3.4.4 Reordenation Attacks

This group of attacks is similar to the Modification group but, in this case, the attacker seeks to change the order of collected records, aiming solutions that depend on chronological ordering of data, a group of which monitoring solutions are part. These attacks can occur during the ordering process of transactions or after they are stored, by manipulating their *timestamp*, as shown in Figure 17.

In the example, the attacker takes advantage of obtaining privileged access to the repository to make changes to the *timestamps* of the events, intending to modify the application execution time or generate inconsistencies in the solution's database. As the reliability of these tools is related to the trust placed in the data ordering process, the success of this type of attack leads to a lack of agreement regarding the order of occurrence of the records, with a direct impact on the solution's functionality. This same

Figure 17 – Reordenation attack example.



Source: Author.

problem, referring to the chronology of events, is also clear when several monitoring tools are running, a scenario in which the lack of synchrony between the solutions' clocks can result in different analyzes and lack of consensus among the actors. Thus, given the importance of guaranteeing the chronological order of events, the solution must implement not only ordering mechanisms, but also ones that seek consensus among the actors, to minimize future conflicts.

3.4.5 Summary of attack model

To define the attack model, the data flow carried out by the event2ledger from the event collection to its storage was analyzed from the perspective of possible attacks to the monitoring solutions. In this sense, each group is represented by a type of attack that illustrates its main objectives and impacts on the solution in case of occurrence. Considering the event2ledger proposal, Table 4 summarizes the attacks, risks, and the adopted defense mechanisms.

Table 4 – Threat model scenarios.

Attack Group	**Risk**	**Defense Mechanism**
Retention attacks	Delay or failure in event registration	Collector authentication and collection window
Modification attacks	Change of events during the collection or after its storage	Collector authentication and guarantee of repository integrity
Insertion attacks	Record creation during collection and MitM attacks	Collector authentication and secure communication
Reordenation attacks	*timestamp* manipulation and Interference in the consensus mechanism	Repository integrity guarantee

Source: Author.

Although event2ledger's proof of concept is useful to prove the solution's theoretical feasibility, the analysis of the attacks presented in the attack model allows to assess whether the proposed solution implements adequate defense mechanisms. Despite being developed for the analysis of event2ledger, the attack model seeks a comprehensive approach to be replicable when analyzing similar solutions. The attacks presented in this model exploit the different stages and characteristics of the

proposed solution, such as its scalability and security during data collection, sending, and storage, to assess its security and feasibility in a production environment.

3.5 CHAPTER CONSIDERATIONS

Through functional and non-functional requirements, it is possible to specify the main characteristics desired in the proposed solution and establish a reference for the search for related works. The proposed objective for the solution is to collect containers and services lifecycle events in a virtualization environment in a non-intrusive way, to use a mechanism that allows the involvement of the actors that compose the scenario in the validation and ordering process of the collected events and store them in a distributed repository. Through a systematic mapping performed in the main academic search engines, five related works, and two open-source tools were identified and evaluated. However, none of them fully meet the specified functional requirements.

Thus, this work proposes a solution called even with two main components: the collectors and a blockchain composed of the three actors involved (Provider, Developer, Customer). In this model, the collectors retrieve the events directly from the local container engine, generating transactions and sending them to blockchain nodes. The blockchain, in turn, is responsible for validating, consensus, and distributed storage of collected events. event2ledger proposes a blockchain structure to integrate security and auditing features into the monitoring solution, ensuring greater confidence in the collection and storage processes and, consequently, less possibility of conflicts between participants. From a security perspective, this work presents an attack model based on the main vulnerabilities that affects monitoring solutions to assess the proposed defense mechanisms.

4 IMPLEMENTATION AND RESULTS

This chapter presents the fundamentals related to the implementation environment and the tools that compose the proposed solution. Then, the infrastructure needed to carry out the proof of concept is described, and the process of configuring and executing the collectors responsible for collecting events, generating, and sending transactions is discussed. The chapter also details the steps for installing and configuring the nodes that compose the blockchain, generating the necessary cryptographic material, and installing the chaincode. With the implementation of the proof of concept, it is possible to assess whether the implemented defense mechanisms are effective against the attacks presented in the attack model proposed in Section 3.4.

4.1 PROOF OF CONCEPT

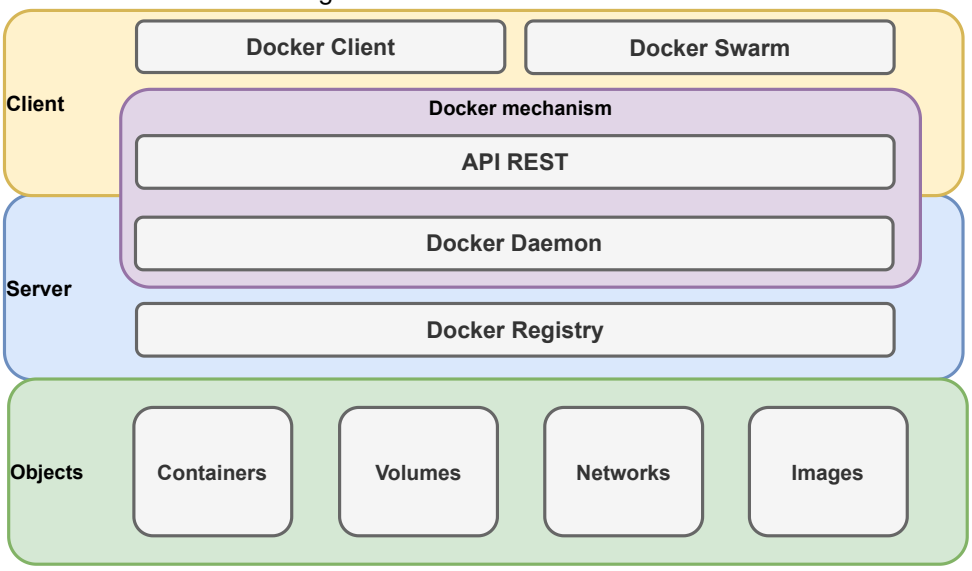
Among the set of tools and technologies available for implementing the solution, those that stand out were chosen, whether from the point of view of adoption or adaptation to the proposed solution. In this sense, the container mechanism adopted was Docker, in an environment orchestrated by the Docker Swarm, a native option of the platform. As for the blockchain, given the need to implement a consortium permission model, the Hyperledger Fabric was chosen, an open-source *framework* developed and maintained by the Linux Foundation.

4.2 DOCKER

Docker is an open-source platform that allows the development, implementation, and execution of applications in containers in an efficient, fast, and light way, separating them from the infrastructure to streamline software delivery (DOCKER.DOCS, 2020). A basic diagram of the Docker architecture is represented in Figure 18.

The model used is client-server, in which the client sends requests or commands to the central component daemon Docker via API Representational State Transfer (REST). The daemon is responsible for building, executing, distributing, and managing the Docker Objects, which, in turn, are the objects created and manipulated by the Docker mechanism. Communication takes place via Hypertext Transfer Protocoll (HTTP), in order to facilitate the connection to daemon. This model has three main elements: the Docker mechanism, the objects, and the record.

Figure 18 – Docker architecture.

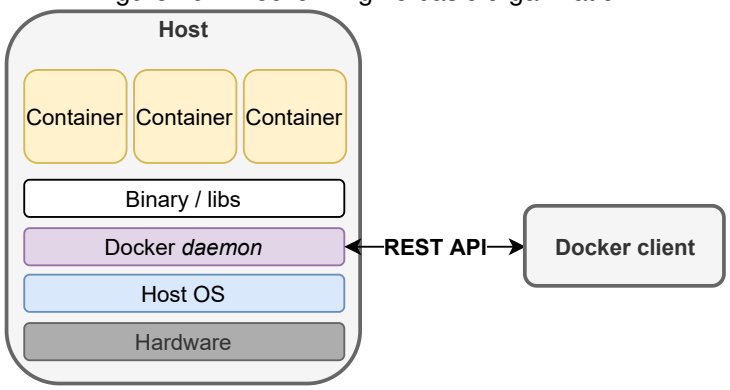


Source: Adapted from (DOCKER.DOCS, 2020)

4.2.1 Docker Engine

The Docker Engine is a client-server application with three main components: Server, API REST and Command Line Interface (CLI). The Server is a long-running program called daemon, which is a central component responsible for responding to requests via API and managing objects. The API REST specifies the interfaces that are to be used to communicate with the daemon. Finally, CLI is the client used to interact with daemon Docker (BUI, 2015). Figure 19 graphically represents the main components of the Docker mechanism.

Figure 19 – Docker Engine basic organization.



Source: (BUI, 2015)

The Docker containers run on the daemon, which is responsible not only for the execution but also for the management of all containers. The CLI uses the API REST to control or interact with the daemon Docker using *scripts* or direct commands, allowing the Docker client to run or not in the same host. The daemon is responsible for creating and managing Docker objects, which are images, containers, networks, and data volumes.

4.2.2 Objects

Objects can be images, containers, networks, volumes, plugins, among others, created and manipulated by the Docker (DOCKER.DOCS, 2020) mechanism.

- **Images**: Are models made up of several overlapping read-only layers, with instructions for creating a specific container. It can be created from a pre-existing image or a base layer, usually a OS, adding additional customizations. The *dockerfile* contains the exact definitions, in the form of script, about the container to be created.
- **Containers**: A container is an executable instance of an image. When created, a layer with write permission is added on top of the existing layers. This layer is responsible for recording the actions of the container in execution, files created, deleted, and modified, in addition to allowing its customization.
- **Volumes**: A volume is an object used for storing data that must persist, i.e., data that should not be discarded after the container has finished running. The process of creating and maintaining the volumes can be done via CLI or API.
- **Networks**: Networks are the mechanism used for communication between containers. The Docker platform has different types of network drivers. The choice of the appropriate driver depends on the environment and the desired isolation level.

Among the objects created and manipulated by the Docker mechanism, containers are the focus of this work. Its life cycle is composed of states that can vary from ”Execution” to ”Excluded”. In a containerized virtualization environment, this transition is performed by the orchestrator, generating life cycle events that can be collected. The Subsection 4.2.4 deals in detail with aspects related to the life cycle of containers in the Docker environment.

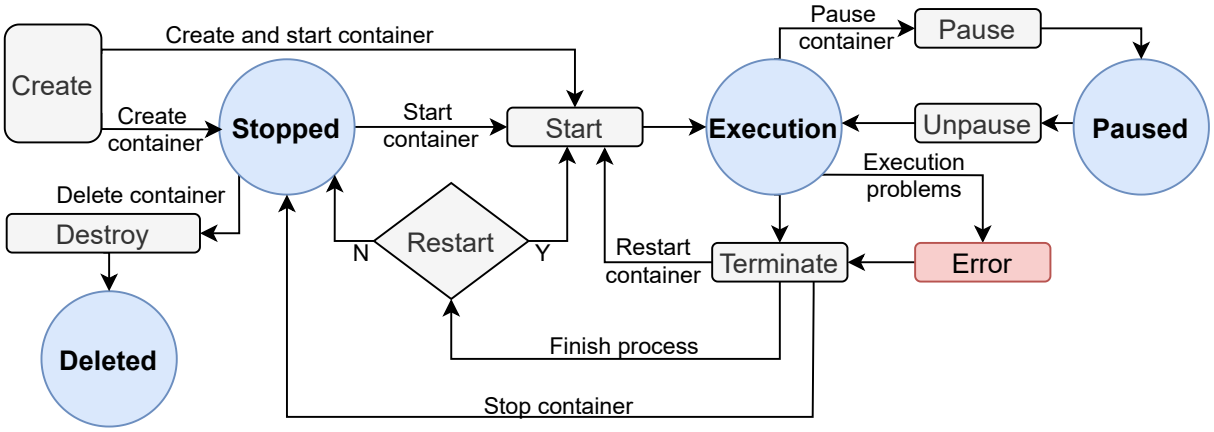
4.2.3 Register

The Registry is a server application responsible for storing and distributing Docker images. This Registry can be public or private. The Docker Hub is a standard public record for the Docker platform. It is also possible to implement a private image registration. Through the use of a private registry is possible to have control over the storage and distribution process of images with commercial or sensitive content, integrating them into the (DOCKER.DOCS, 2020) development flow. The Registry can store base images, used for the creation of containers, as well as images of ready-to-use applications, including databases, variables, content, services, among others.

4.2.4 Docker container life cycle

Typically, the life cycle of Docker containers is composed of the states: Execution, Paused, Stopped, or Deleted. Figure 20 represents the possible transitions between the life cycle phases of Docker containers.

Figure 20 – Container life cycle.



Source: Adapted from (MOUAT, 2015)

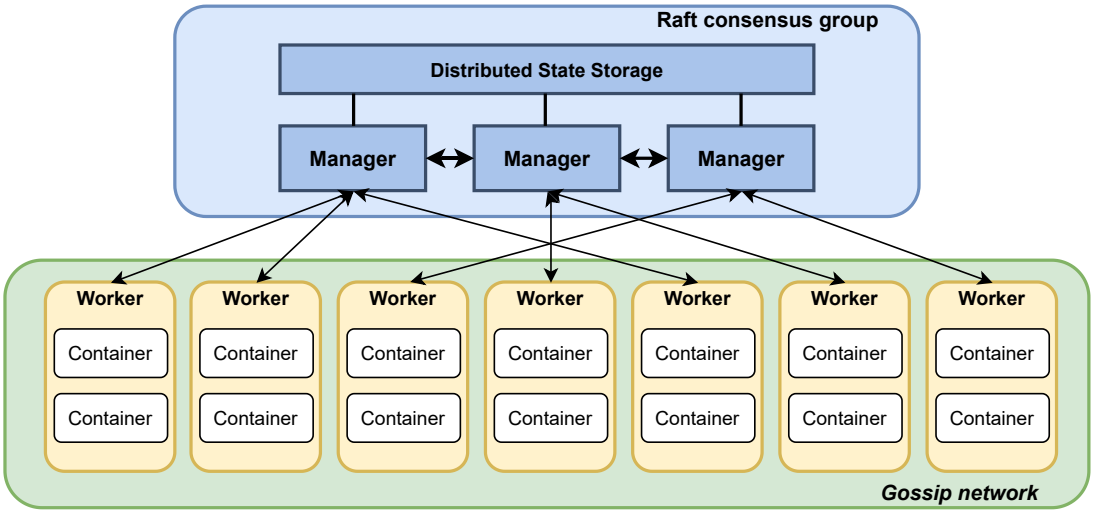
The transition actions between the stages of the life cycle can be performed via CLI or Orchestrator, with the Docker mechanism being the component responsible for managing the life cycle of the containers. Each of the objects managed by Docker has a set of events related to its life cycle. Among the events available for the container object, this work focuses on the collection of creation, pause, finalization, and destruction events, specifically: create, destroy, pause, unpause, start, and stop.

4.3 DOCKER SWARM

Docker Swarm is a tool in the Docker ecosystem that allows the creation of clusters composed of one or more Docker hosts. Through it is possible to manage the life cycle of containers in a distributed Docker environment. In Docker Swarm, there are two types of nodes: Manager and Worker, each with different functions. Figure 21 shows the basic architecture of Docker Swarm.

In this model, Manager nodes are responsible for maintaining the state of the cluster, scheduling the services, and providing the API. For the services to be able to locate and communicate, a discovery service is required, responsible for storing information about the location of each one of them. The consistency of the state of the cluster and the services in execution is maintained through the implementation of a distributed consensus mechanism called Raft, whose objective is the maintenance of replicated logs (ONGARO; OUSTERHOUT, 2014), and is described in detail in subsection 4.4.2.1. It is through a consensus algorithm that a set of machines can work as a coherent group, capable of supporting the loss of some nodes without the loss of

Figure 21 – Docker Swarm architecture.



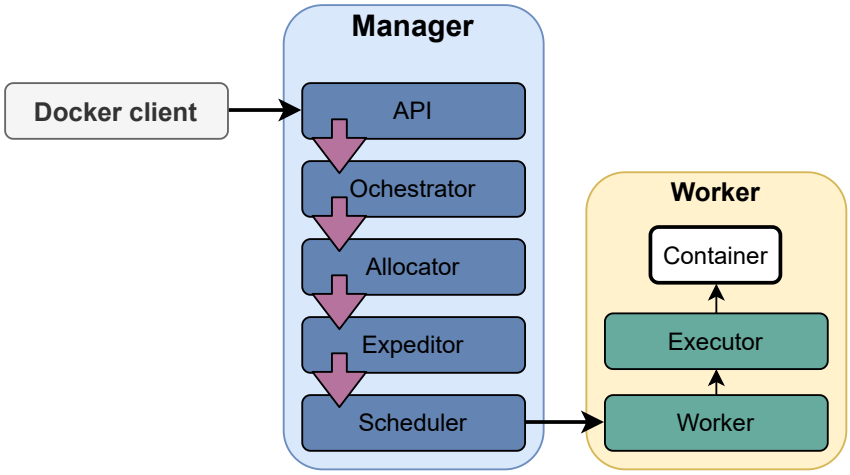
Source: (DOCKER.DOCS, 2020)

information. To make use of the fault tolerance features provided by the Docker Swarm orchestration platform, it is recommended to use an odd number of Manager nodes, according to the predefined high availability requirements. In general, a cluster of Manager nodes is composed of *N* nodes is able to support the loss of, at most, *(N-1) / 2* nodes (DOCKER.SWARM, 2020). However, the official documentation for the Docker Swarm tool recommends a maximum of seven Manager nodes, noting that adding more Manager nodes does not imply greater performance. The Worker nodes have the task of executing the tasks received from the Manager nodes.

Implementing an image of a container with Docker Swarm is accomplished by creating a service. When a service is created, it specifies which image to use, which commands will be executed within the container, in addition to defining variables such as the number of replicas, available network and storage resources, reservation and processing, and memory usage limit, among others. After defining the service, the Manager node assumes the specified variables as the desired state for the service, and Docker Swarm will work to maintain the defined state. The service is then scheduled to run on one or more Worker nodes, according to the replica definitions applied to it, and is executed as a task. In the Docker Swarm model, a task invokes only one container, being analogous to a vacancy in which a container can be allocated by the Scheduler. In this way, if a node becomes unavailable, then its tasks are rescheduled to other nodes to maintain the level of service availability.

Task scheduling is how the orchestrator seeks to reach the desired state defined by the creation or update of a service. A task is a unidirectional mechanism, which progresses through the stages of its life cycle according to the predefined desired state (DOCKER.DOCS, 2020). Figure 22 shows the process of receiving and allocating service calls by Docker Swarm.

Figure 22 – Docker Swarm service call.



Source: Adapted from (DOCKER.DOCS, 2020)

The API of the Manager node receives commands from the Docker client, passing them to the Orchestrator. This, in turn, creates the tasks according to the commands received via API. The Allocator is the component responsible for allocating Internet Protocol (IP) addresses for the created tasks. After assigning the addresses, the Dispatcher assigns the tasks to the Worker nodes. Finally, the Scheduler is responsible for informing the Worker node when to perform the tasks. The Worker node has two main components: the Worker component, responsible for verifying the tasks assigned to the node, and the Executor component, responsible for performing the tasks assigned to the node (DOCKER.DOCS, 2020). When the tasks of a service cannot be performed by any of the Swarm nodes, their state is defined as pending. It can occur if all nodes are paused, due to insufficient computational resources or any other predefined variable that cannot be reached.

In the Docker Swarm model, there are two types of service implementation: replicated and global. A service of the replicated type has, by default, a specific number of identical tasks to be performed. A global service, on the other hand, performs a certain task on all nodes of Swarm (DOCKER.DOCS, 2020). In this way, whenever a new node is added, the orchestrator creates, for every global service, a new task, and the scheduler assigns it to the new node. Due to this characteristic, this model is used in deployments of monitoring agents, causing an instance of the agent to run on every new node in the network.

Docker allows the collection, at runtime, to log container life cycle events by communicating with the Docker engine. Logs are recorded information regarding running containers, such as processing and memory consumption. The events are information related to the life cycle of containers, such as creation, finalization, copying, and destruction (DOCKER.DOCS, 2020). The main purposes of monitoring a cloud computing environment are the efficient use of computing resources, pricing and per-

formance monitoring (SYED et al., 2017). In addition, monitoring requires a chronological sequencing of the collected data. Considering the scope of this work, it is possible, through container events, to track the execution of containers in a virtualization environment.

4.4 BLOCKCHAIN

The blockchain is a secure, shared and distributed ledger that facilitates the process of recording and tracking resources without the need to trust a central authority, allowing two parties to communicate and exchange resources over a peer-to-peer network where decisions are made by the majority, not a single entity (Salman et al., 2019). This ledger is composed of cryptographically signed transactions grouped in blocks. In a blockchain, each block contains, in the form of *hash*, a representation of the previous block, creating a cryptographic binding validated using a consensus mechanism. This arrangement ensures that the blockchain transaction history cannot be changed or deleted without invalidating the string of *hashes* (TINU, 2018). As new blocks are added to the chain, it becomes more difficult to change old blocks (YAGA et al., 2018). These characteristics make the blockchain a suitable technology to meet the functional requirements defined in Section 3.1.

Different types of blockchains available differ in terms of their permission model and consensus mechanism. The combination of these variables results in blockchain models with distinct applications. Thus, it is necessary to analyze the characteristics of each of them to identify the most suitable blockchain model and consensus mechanism for the proposed solution.

4.4.1 Permissioning model

The blockchain network can be categorized according to its permission model, being permissionless or permissioned. In the permissionless model, any node can publish a new block on the blockchain network. In the permissioned model, only authorized nodes can publish blocks (YAGA et al., 2018). Based on the adopted permission model, blockchains are categorized into public, consortium, or private (MIERS et al., 2019a):

- In a public blockchain, everyone can read, send or validate transactions, in addition to participating in the distributed consensus process, which is considered a fully decentralized model.
- A consortium blockchain, in turn, is composed of two or more partner institutions that can change the rules according to their needs. Consensus is achieved

through a process carried out by a specific group of participants and is thus partially decentralized.

- A private blockchain, on the other hand, is used in single organization models that can change the way it works according to its interests and needs. It is the most centralized model, where the consensus process can be achieved most simply.

Each of these categories has a distinct application based on a set of blockchain characteristics. Table 5 presents some criteria for comparing the main characteristics of the three blockchain types (MIERS et al., 2019a).

Table 5 – Comparison between blockchain access models.

Characteristics	Public Blockchain	Consortium Blockchain	Private Blockchain
Distributed consensus	All nodes	Selected nodes	Selected nodes
Verification permission	Public	Restricted	Restricted
Immutability level	High	Medium	Low
Centralization	Descentralized	Partial	Centralized
Consensus process	All nodes	Selected nodes	Selected nodes

Source: Author.

Distributed consensus defines whether all nodes can participate in the consensus process or just predetermined nodes. The verification permission can vary between public or restricted, with the identity of the participants also being able to be anonymous in public blockchains or known, in the consortium and private models (TINU, 2018). Although, by characteristic, the data stored in public blockchains are difficult to modify due to its decentralization, in consortium or private model it can be performed in a reasonably easy way. As for the degree of centralization, models can range from fully centralized in the private blockchain model to decentralized in the public model. Finally, the consensus process defines whether any entity can participate in the process or only pre-selected entities (MIERS et al., 2019a). In this way and based on these characteristics, it is possible to define the consortium blockchain as the most suitable model for the solution proposed by this work.

4.4.2 Consensus Mechanisms

Due to its asynchronous and decentralized nature, one of the main aspects of blockchain technology is determining who will publish the next block. When a node joins a blockchain it agrees with the initial state of the system, defined in the Genesis block. Every following block must be valid and also possible to be validated, independently, by any node in the network. Combining the initial state with the ability to check each block since then, users can agree on the current state of the blockchain (YAGA et al., 2018). However, each node can have a different view of the blockchain state at a given time. To ensure the convergence of these points of view, the blockchain uses a consensus

mechanism that allows distributed or decentralized networks to make unanimous decisions when necessary (SANKAR; SINDHU; SETHUMADHAVAN, 2017). This mechanism creates a consistent system where all nodes agree on the order of blocks and their contents.

Some of these mechanisms, known as Bizantine Fault Tolerant (BFT), rely on Byzantine Fault Tolerance, guaranteeing fault tolerance and malicious behavior for up to a third of the total of nodes (SANKAR; SINDHU; SETHUMADHAVAN, 2017). Other mechanisms, called Crash Fault Tolerant (CFT), offer tolerance to faults but not to malicious behavior of the nodes. However, it requires less computational resources for its implementation if compared to BFT mechanisms. The choice of the type of consensus mechanism must take into account the reliability of those involved in the process, as well as the feasibility of implementing the mechanism in the proposed environment. The scenario in question adopts a consortium blockchain composed of three different actors with the same node amount. From the point of view of use in consortium blockchain, (MIERS et al., 2019a) points out that, in general, the Practical Byzantine Fault Tolerance (PBFT) and Delegated Proof of Stake (DPoS) consensus mechanisms are the most used. In turn, (ZHENG et al., 2018) cites the PBFT, Tendermint and DPoS mechanisms as preferentially used in this blockchain model. Already (DIB et al., 2018) lists, among the main consensus mechanisms for consortium blockchains, the PBFT, Tendermint and Proof of Elapsed Time (PoET). In addition to the mentioned models, considering the existing consortium blockchain models and *frameworks*, the Raft consensus mechanism stands out for being the standard adopted by the open-source solution Hyperledger Fabric, developed by the Linux Foundation, and made available by the largest providers of cloud. These mechanisms, identified as potential candidates for use in the proposed solution, have the following specific characteristics:

- The PBFT is a widely tested Byzantine fault-resistant consensus mechanism (DIB et al., 2018). In this model, nodes operate in rounds. In each round, a primary node is selected and is responsible for inserting the next block in the chain. This process is divided into three phases, and it is necessary to receive the vote of 2/3 of the total number of us to advance from one phase to another (MIERS et al., 2019a). This mechanism requires low computational effort and was used by Hyperledger Fabric in its early versions.

- Tendermint is a low computational effort consensus algorithm developed with a specific focus on blockchain consortium. It is similar to PBFT, in which blocks are added by rounds of voting, requiring the approval of at least 2/3 of the nodes. It has features that allow you to audit and identify malicious nodes (BUCHMAN, 2016). However, in this model nodes have coins that must be blocked to become

validators (ZHENG et al., 2018). From a performance point of view, compared to PBFT, the commit time of a block in Tendermint is five times longer, and the maximum number of transactions per second, five times smaller (DIB et al., 2018).

- PoET takes a different approach in choosing the leader node responsible for producing the new block. To do so, it uses the Software Guard Extension (SGX) resource made available by Intel processors, as of 2015, with which potential validator nodes sign an entry message that, if accepted, enables them to participate in the leader selection process (HYPERLEDGER, 2020c). However, the use of this mechanism necessarily implies the use of Intel SGX or a similar specific platform that provides a safe execution environment (Trusted Execution Environment (TEE)). PoET is the mechanism used by Hyperledger Sawtooth and requires low computational effort to reach consensus.

- The DPoS elects a list of nodes that will have the opportunity to participate in the generation of the block containing new transactions and add them to the blockchain. In this way, fewer nodes are needed to generate and validate the blocks, speeding up process execution. This model, however, tends to choose validator nodes with low variation, in addition to presenting difficulties in dealing with malicious nodes (Yang et al., 2019). DPoS is the consensus mechanism used by Bitshares. The computational effort to obtain consensus in DPoS is greater than in other candidate mechanisms.

- Raft is the CFT consensus engine used by the Hyperledger Fabric, the most popular *framework* of the Hyperledger umbrellas. Its implementation is simpler and requires less computational resources if compared to BFT mechanisms. The Raft operates in rounds, through a leader election process, responsible for replicating the operations to the other nodes (ONGARO; OUSTERHOUT, 2014).

After analyzing the characteristics presented and the demands of the proposed solution, it was possible to identify the feasibility of the candidate consensus mechanisms. The PoET mechanism was discarded for demanding an TEE for its implementation. In the DPoS model, despite being a usual solution in consortium blockchains, there is still some demand for computational resources to reach consensus. Tendermint and PBFT are similar models and were considered as viable implementation alternatives. However, compared to PBFT, Tendermint still lacks more implementations and studies. The Raft mechanism, in turn, even though it is not an BFT model, was considered a viable alternative because, in addition to having a validation process that requires less computational resources, it is easy to implement in consortium blockchain environments. With the functional requirements established, the PBFT and Raft mechanisms

were considered consensus mechanisms capable of being implemented in the proposed model. Thus, considering the adequacy of the implementation environment and the solution's functional requirements, Raft was chosen as the consensus mechanism to be used.

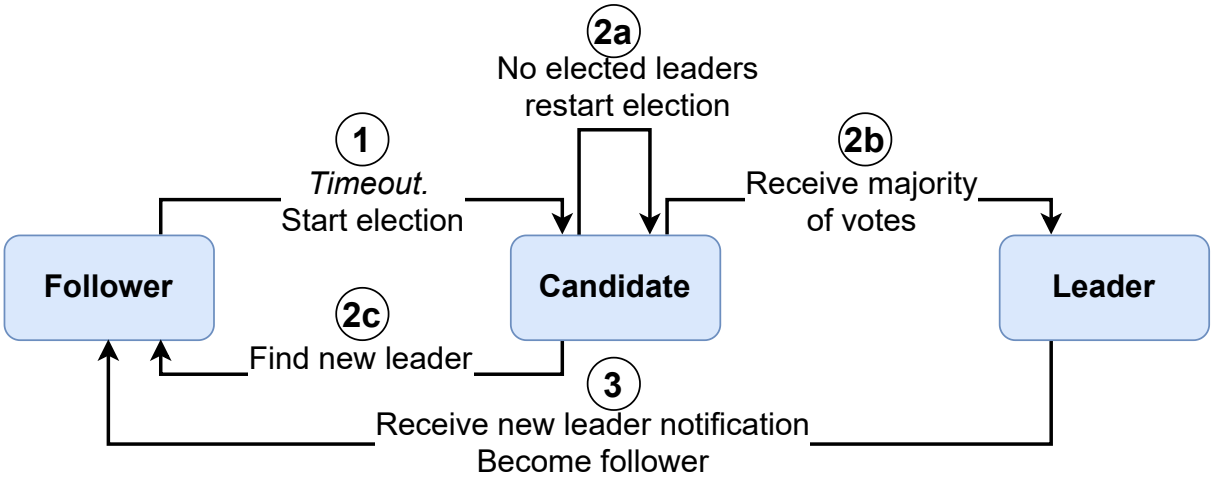
4.4.2.1 Raft

As defined in (ONGARO; OUSTERHOUT, 2014), Raft is a consensus algorithm developed for managing replicated *logs*. It is a mechanism equivalent to PAXOS but with a structure aimed at simplifying the process. In this model, Raft separates the main elements for consensus (*i.e.*, leader election, *log* replication, and security) using a high degree of coherence to reduce the number of states to be considered. Raft's architecture, although similar to some other implementations of consensus mechanisms, has some specific characteristics:

- **Strong leader:** The leadership model used in Raft is stronger than in other mechanisms, with *log* entries flowing only from Leader to Member, simplifying *log* management and facilitating understanding;
- **Leader election:** Raft uses random timers for leader election. This way, a small increment of time in the already existing *heartbeats* can easily resolve conflicts; and
- **Member changes:** The raft has a mechanism that allows you to change the set of nodes in the cluster so that most nodes in two different configurations overlap during changes. This feature allows the cluster to continue to function during changes to its settings.

To obtain consensus the Raft mechanism initially elects a leader node responsible for managing the replicated *log*. This leader receives client requests and replicates them to the other cluster nodes, informing when the updates must be applied in their respective machine states. In Raft's original work, (ONGARO; OUSTERHOUT, 2014) breaks down the consensus-building process into three independent sub-problems: Leader election, *log* replication, and Security. The leader election process is essential as it is the leader who defines the node responsible for obtaining customer requests and sending them to all nodes in the cluster. Figure 23 represents the possible states of a node and its transitions.

Figure 23 – Raft node states and transitions.

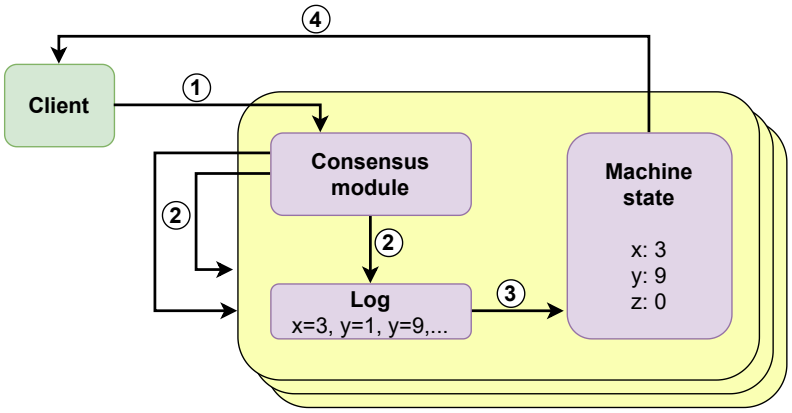


Source: (ONGARO; OUSTERHOUT, 2014)

This model operates based on a leader node. This way, when starting a Raft cluster, all nodes that make it up are in the follower state. Periodically, the leader node sends *heartbeats* to all followers to maintain its leadership. When not receiving a *heartbeat* from a leader, the follower nodes apply for and start electing a new leader (1). Each node can only vote for one of the candidates. In some situations, voting can result in a tie. In these cases, there is no leader election, and a new round of voting takes place (2a). When one of the candidates receives the votes of the majority of the cluster nodes, it becomes the new leader (2b), sending a *heartbeat* to the other nodes to establish the leadership and prevent further rounds of voting. After the election, a leader typically operates until it fails, when a new voting process begins.

With the elected leader, the other nodes become followers and this one starts to respond to customer requests (2c). Each request contains an operation to be performed and applied to the replicated state. To do this, the leader adds the operation to his *log*, sending calls like *appendEntry* to the other nodes. The leader is responsible for deciding when it is safe to apply the *log* entry to the machine state, a process known as *commit*. For this, the operation must be replicated to most nodes. After the operation commit, Raft guarantees that its execution by all available nodes. When the operation is replicated the leader applies it to the machine state, returning the result to the client. In cases of eventual communication problems or slowness between the leader and the followers, the leader keeps trying indefinitely until all the followers store the log. Figure 24 represents the adopted machine state replication architecture.

Figure 24 – Raft State Machine Replication Architecture.



Source: Adapted from (ONGARO; OUSTERHOUT, 2014)

In this model, the leader node receives, from the client, the operation to be performed (1). The consensus module is responsible for maintaining the consistency of the replicated *log*. It receives the operations and adds to the *log* of the leader node communicating with the consensus modules of the other nodes (2) to guarantee that all nodes have the same operations to be executed in the same order, resulting in the same machine state (3). Finally, the leader node returns the result of the operation to the client (4).

Raft is the default consensus engine adopted by Hyperledger Fabric, a *framework* that is part of the Hyperledger Umbrellas project, developed with a focus on private and syndicated blockchain implementations, and available from major cloud computing providers. It can be implemented in a virtualization environment in Docker containers, facilitating the blockchain integration with the proposed solution. Thus, from the analysis of its characteristics, it is possible to assess the adequacy of the Hyperledger model to the established functional requirements.

4.5 HYPERLEDGER

The Hyperledger Project is an initiative by the Linux Foundation to develop an open-source blockchain development ecosystem, that aims to create an environment in which communities of *software* developers and companies come together and coordinate to build blockchain structures. The Hyperledger model is a cross-platform, modular and open source project. This approach presents, as differentials, features such as extensibility, flexibility, and the ability to modify any component without affecting the system (HYPERLEDGER.ARCHITECTURE, 2017). Hyperledger is an umbrella project, under which five distinct *frameworks* are being developed (DHILLON; METCALF; HOOPER, 2017):

- **Sawtooth**: Developed by Intel to create an open-source blockchain, Hyperledger

Sawtooth is a modular platform for developing, implementing, and running Distributed Ledger Technology (DLT). It adopts a consensus mechanism known as PoET, which aims to obtain consensus with minimal computational effort. It is a *framework* that allows the implementation of permissioned and permissionless solutions.

- **Fabric**: This *framework* aims to allow the development of business applications with a modular architecture, enabling the adoption of different consensus mechanisms and unique membership services.
- **Iroha**: Set of libraries and components that allow the implementation of DLT in existing infrastructures.
- **Burrow**: Permissioned blockchain to smart contracts execution similar to Ethereum Virtual Machine (EVM). Allows the execution of smart contracts in multiple blockchains compatible with each other but running in different domains.
- **Indy**: It is a Software Development Kit (SDK) for Hyperledger that offers components that allow you to add new features and functionalities for decentralized identity management.

Thus, there are distinct Hyperledger blockchain projects, each one having characteristics for a specific solution model. However, all Hyperledger projects follow a development model which includes a modular and extensive approach, interoperability, and emphasis on secure solutions, with a *token* independent approach and no native cryptocurrency, in addition to the use of API.

4.5.1 Hyperledger Fabric

Hyperledger Fabric is a modular blockchain architecture that allows the connection of components such as consensus engine, membership services, and transaction functions. This feature allows effective customization of the platform according to the needs of the environment. Thus, when implemented in an environment composed of only one company, for example, the use of a CFT consensus mechanism may be more suitable than a BFT mechanism. On the other hand, in decentralized multi-organizational environments, a BFT mechanism may be needed (HYPERLEDGER, 2020a). In addition to the consensus engine, other Hyperledger Fabric components are modular and configurable:

- *Ordering service*: Responsible for establishing a consensus on the order of transactions and sending them to *peers*.

- *Membership service provider*: Responsible for associating entities that make up the network to cryptographic identities.
- *Peer-to-peer gossip service*:Responsible for disseminating the blocks to all nodes.
- *Chaincode*: In Hyperledger Fabric smart contracts are also known as *chaincode* and can be developed in Go, Javascript, and eventually another language such as Java. In this model, there are two types of *chaincode*: *system chaincode* and *application chaincode*. The *system chaincode* typically handles system-related transactions, such as lifecycle management and policy settings. The *application chaincode* manages the state of the *ledger*, including data records.

Additionally, the platform also supports many database management systems and allows many validation and endorsement policies configurations. In Hyperledger Fabric, the blockchain is composed of nodes belonging to the organizations that make up the consortium, in addition to one or more nodes of the *orderer* type, responsible for ordering the transactions. Only the nodes that are in the same channel can communicate with each other.

4.5.2 Channels and components

The model proposed by Hyperledger Fabric allows organizations to participate in multiple independent blockchain networks through channels. Every transaction must be executed in a channel where all participants are authenticated and authorized to carry out transactions. The channel offers the sharing of infrastructure maintaining the privacy of data and communication and is composed of member nodes (*peer*) belonging to the participating organizations, anchor nodes (*anchor peer)*, ordering nodes of the transactions (*order peer)*, *ledger* and *chaincode*.

All organizations that compose the channel must have at least one *anchor peer*. The *anchor peer* are responsible for communicating with the *orderer peer*, obtaining the blocks containing the transactions, and disseminating them to the other nodes of their organizations. Although they can belong to multiple channels and have several *ledger*, the channel data is private and cannot travel between them. The *orderer peer* are responsible for ordering transactions through a consensus mechanism, assembling the blocks, and delivering them to the *anchor peer* (HYPERLEDGER, 2020a).

4.5.3 Transaction architecture

A transaction, or *Transaction proposal*, is the result of calling a *chaincode* to collect endorsements, ordered globally in the context of a channel and validated by the *committing peers* as part of a block, before updating the *ledger*. Each transaction

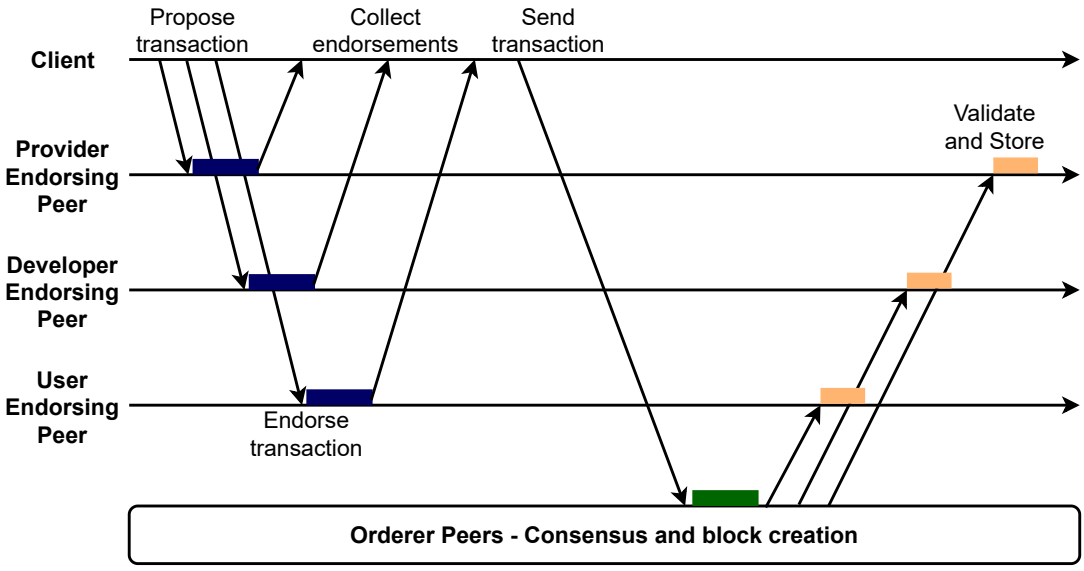
contains an array of actions that represent different steps in the execution of a transaction. These steps are processed atomically, which means that if any fail, the entire transaction is marked as rejected (HYPERLEDGER, 2021).

Hyperledger Fabric implements, in opposition to the traditional two-step transaction architecture model (*i.e.*, *order-validate*), a new three-step model known as *execute-order-validate*. The first step executes the transaction, checks its accuracy, and endorses it. The second step orders transactions using a consensus mechanism. The third step validates transactions through an application-specific validation policy, before adding it to the *ledger*. The validation policy specifies which and how many nodes should validate the execution of *chaincode*.

This architecture adds features to deal with challenges related to scalability, performance, and confidentiality existing in the traditional *order-validate* model. One can imagine, as an example, a *chaincode* whose execution is an infinite *loop*. In an *order-execute* architecture, this *chaincode* would have a critical consequence. The *execute-order-validate* model is able to identify this problem in the first execution step when the nodes must endorse the transactions that will be submitted for validation. In this model, the transaction would not be endorsed and would be discarded before being passed on to the *orderer* and endorsement nodes.

To perform the execution and interaction with *chaincodes*, Hyperledger Fabric makes use of transactions. There are two types of transactions: *Invoke* and *Query* (DHILLON; METCALF; HOOPER, 2017). The *invoke* transaction executes a *chaincode* on the blockchain, while the *query* transaction executes a query. Through a *invoke* transaction it is possible for a client to execute a specific function contained in a *chaincode*. Figure 25 illustrates how applications interact with nodes to access the *ledger*.

Figure 25 – Hyperledger Fabric transaction flow.



Source: Adapted from (PAJOOH et al., 2021).

In this model, the process starts with the transaction proposal sent by the client to all blockchain endorsement nodes. Then, these nodes execute the *chaincode* function called, performing the predefined authentication and authorization procedures. Transactions are then signed and returned to the customer. When the client receives the required number of endorsements (defined in the blockchain's endorsement policies), the transaction is sent to the transaction ordering service (*orderer peers*), responsible for obtaining the consensus and order of transactions. After ordering, transactions are sent to blockchain nodes, which check them and store the results in the *ledger*.

4.5.4 Transaction validation

It is important to differentiate the transaction validation process from the data validation process. Transaction validation is a result of the blockchain transaction architecture. During this process, a series of characteristics inherent to the transaction sent are verified, such as the permissions referring to the key used to sign the transaction, the transaction ID, the necessary endorsements, among others. Table 6 lists the possible transaction validation codes.

Table 6 – Transaction validation codes.

Validation Code	Description	Validation Code	Description
0	VALID	**13**	UNKNOWN_TX_TYPE
1	NIL_ENVELOPE	**14**	TARGET_CHAIN_NOT_FOUND
2	BAD_PAYLOAD	**15**	MARSHAL_TX_ERROR
3	BAD_COMMON_HEADER	**16**	NIL_TXACTION
4	BAD_CREATOR_SIGNATURE	**17**	EXPIRED_CHAINCODE
5	INVALID_ENDORSER_TRANSACTION	**18**	CHAINCODE_VERSION_CONFLICT
6	INVALID_CONFIG_TRANSACTION	**19**	BAD_HEADER_EXTENSION
7	UNSUPPORTED_TX_PAYLOAD	**20**	BAD_CHANNEL_HEADER
8	BAD_PROPOSAL_TXID	**21**	BAD_RESPONSE_PAYLOAD
9	DUPLICATE_TXID	**22**	BAD_RWSET
10	ENDORSEMENT_POLICY_FAILURE	**23**	ILLEGAL_WRITESET
11	MVCC_READ_CONFLICT	**255**	INVALID_OTHER_REASON
12	PHANTOM_READ_CONFLICT		

Source: Author.

The codes represented in the table indicate the success (*i.e.*, Code 0) or reason for the failure of the transaction validation process. These processes validate characteristics such as transaction construction, sender signature, *chaincode* execution, among others. Data validation, in turn, is possible through its *hash*, included in each transaction. This feature is associated with cryptographic chaining and guarantees the traceability of possible changes.

4.5.5 Hyperledger Fabric binaries

Hyperledger Fabric has a set of binaries responsible for the main functionalities of the platform. These binaries are needed during the blockchain creation and configuration process, and can also be used in the development of applications that interact with Hyperledger Fabric. Thus, the binaries and their functionalities are:

- *Configtxgen*: Responsible for generating network artifacts, such as genesis.block and channel.tx;
- *Configtxlator*: Responsible for generating and configuring the communication channel;
- *Cryptogen*: Responsible for generating the cryptographic material;
- *Discovery*: Command line client for discovery service;
- *Idemixgen*: Utility for generating keys to be used with Membership Service Provider (MSP);
- *Oderer*: Responsible for interacting with the ordering node;
- *Peer*: Responsible for interacting with the validation node; and
- *Fabric-ca-client*: Client for creating and registering users.

4.6 IMPLEMENTATION

The proof-of-concept implementation consists of two steps: blockchain installation and configuration and collector installation and configuration. It is during this phase that the blockchain nodes and the communication channel are created and started.

It is also at this moment that the permissions policies and network participants are defined. The chaincode installation and update process requires the signature of all channel participants, which prevents its unilateral manipulation. In this step, the consensus mechanism to be used during the chronological ordering process of the transactions containing the events is also defined. Also during blockchain implementation, the certificates used by collectors and blockchain nodes in the signature of transactions and endorsements are generated. Collectors, in turn, were developed as *scripts* bash and use CLI to interact with Docker during event collection and with blockchain nodes while sending transactions.

4.6.1 Implementation environment

The proof of concept was implemented in an OpenStack cloud, in a virtualization environment in Docker containers, using the Docker Swarm Orchestrator. The cluster that compose the Docker Swarm has three nodes: one *manager* and two *workers*. All blockchain nodes, in turn, were implemented in containers and executed only in the *manager* node of the Docker Swarm. Table 7 presents the computational resources allocated to the nodes.

Table 7 – Computational resources allocation

Node	vCPU	vRAM	Disk
Manager	4	8 GB	20 GB
Worker 1	1	2 GB	20 GB
Worker 2	1	2 GB	20 GB

Source: Author.

The computational resources allocated to the nodes are sufficient for the execution of the proposed infrastructure, such as orchestrator, blockchain, and event2ledger collectors, in addition to containers, applications, and services, so that it is possible to carry out the proof of concept and simulation of events for collection. The three nodes of the cluster run, as a base, a GNU/Linux Ubuntu Desktop 20.04 distribution.

4.6.2 Hyperledger Fabric Blockchain

For validation and storage of collected events, the solution uses a containerized Hyperledger Fabric consortium blockchains. Thus, the nodes that make up the blockchain (*i.e.*, endorsement, ordering, storage, and Certificate Authority (CA)) are containers created and executed in the Docker environment to be monitored. The use of Hyperledger Fabric requires the installation of a set of prerequisites, listed in (HYPERLEDGER, 2020b), to be able to implement the blockchain, generate cryptographic keys, install and execute the *chaincode*.

As described in the Section 3.3, the blockchain implemented in this work is a consortium model composed of three actors. In this model, each actor has a set of nodes, with specific functions for the validation process and transaction storage, and also a CA (one for each participating actor) and the nodes responsible for executing the consensus mechanism and order transactions. In this way, the blockchain implementation creates and executes the following containers in the cluster's *manager* node:

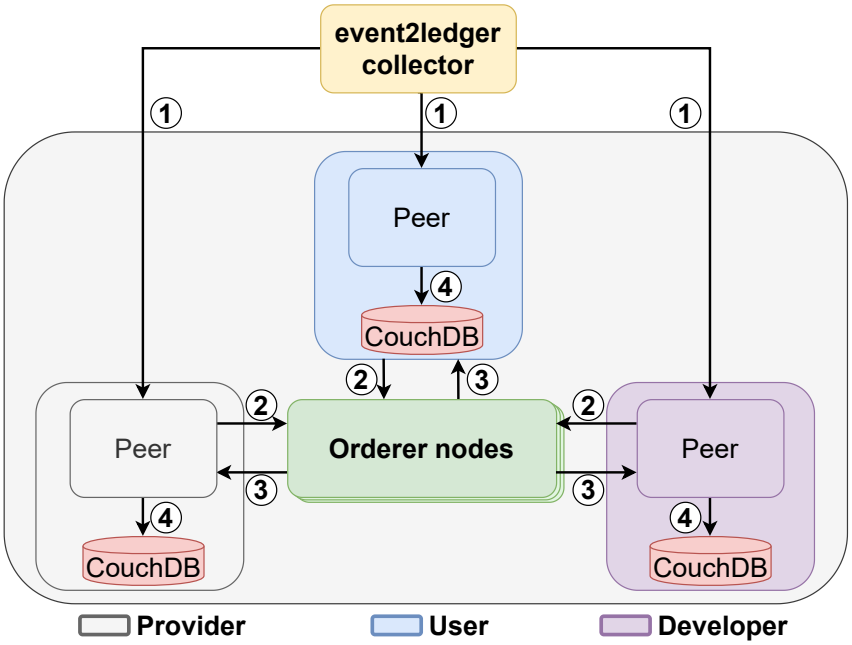
- Provider:
  - ca.provedor.labenv.com
  - peer0.provedor.labenv.com
- Deeloper:
  - ca.developer.labenv.com
  - peer0.developer.labenv.com
- User:
  - ca.user.labenv.com
  - peer0.user.labenv.com
- *Ordering nodes*:

  - orderer.labenv.com
  - orderer2.labenv.com
  - orderer3.labenv.com

- Storage:
  - couchdb0.labenv.com
  - couchdb1.labenv.com
  - couchdb2.labenv.com

The set of nodes presented and implemented in the proof of concept is the minimum requisites to execute the blockchain implementing the Raft consensus engine and distributed storage with the fault tolerance of one of its nodes. In this model, the node (*peer0*) is responsible for executing, endorsing, validating, and storing the transactions received. Actors also have a CA, responsible for generating their cryptographic keys. The blockchain also has three ordering nodes responsible for consensus on the order of transactions and distribution of blocks and three nodes for storage (CouchDB). Figure 26 presents an overview of the implemented Hyperledger Fabric blockchain and its respective transaction flow.

Figure 26 – Proposed blockchain transaction flow.



Source: Author.

When the event2ledger collector receives an event, it generates a transaction signed with its private key, issued by one of the authorized CA and forwards it to the Hyperledger Fabric endorsement nodes (1). Endorsement nodes perform the requested chaincode function and sign the received transaction until the minimum number of signatures required in the endorsement policies is reached. At this point, the transaction is sent to the sort nodes (2). These nodes use the Raft consensus mechanism to ensure

there is consensus on the chronological order of transactions. After ordered, a block containing the transactions is created and sent to all blockchain nodes (3) that will validate them based on the policies defined in the blockchain creation and store the result in the *ledger* (CouchDB) (4).

The described process takes place within a secure communication channel, created during the blockchain implementation, to which the actors' nodes that make up the network are connected. Then the *chaincode* is installed on the endorsement nodes, containing a set of functions that allow you to create and check events stored in the blockchain. Table 8 presents the main configuration parameters applied to the blockchain implemented in the proof of concept.

Table 8 – Blockchain Configuration Parameters.

Parâmetro	**Valor**
Number of channels	1
Chaincode language	Go
***Ledger* model**	Couchdb
Endorsement policy	Major Endorsement
Transactions per block	50
Block *Timeout*	3 s

Source: Author.

Although it is possible for a node to belong to more than one channel, the model proposed in this work includes the installation of only one channel in each node. The proposed solution uses CouchDB as the *ledger* of the blockchain, allowing the elaboration of complex queries to the database. The endorsement policy adopted requires transactions to be validated by the majority of participants (*i.e.*, at least two). As for the settings referring to blocks, the implemented model adopts blocks containing a maximum number of 50 transactions, with a *timeout* of 3 seconds (*i.e.*, after 3 seconds, the block is sent even if it does not contain 50 transactions).

4.6.3 *Chaincode*

In the Hyperledger Fabric model, the *smartcontract* is also called *chaincode*, and can be developed in Javascript, Go, or Java. As described in (HYPERLEDGER, 2020a), an intelligent contract defines the transaction logic that controls the lifecycle of a business object contained in the global state, being wrapped in a *chaincode* and deployed to a network blockchain.

The *chaincode* used in the proposed solution was developed in Go language and implement the Event data type to facilitate the manipulation and storage of received data, and was defined according to the event fields and two additional fields, referring to the transaction ID and the certificate *CommonName* used to send the transaction. In addition, the *chaincode* has a set of functions designed to allow the insertion of new

events, as well as the query and validation of events already stored. Thus, the functions present in *chaincode* are:

- **createEvent**: The createEvent function is called always that a new event is collected and its execution occurs after the private key and permission validation. Its main function is to store the event data under a container ID orientation. The container ID is used as an index to store the events and their respective data to facilitate the search of a specific event or the event history of a certain container. In the createEvent request, in addition to the function name, the collector also sends the event data and informs the index used in the repository (*i.e.*, container ID). Upon receiving the call, the function checks if the number of parameters is correct and then also collects and stores the *CommonName* present in the certificate used to sign the transaction in addition to the transaction ID.
- **queryAllEvents**: The queryAllEvents function aims to return all events stored in the *ledger*. This way, this function does not demand any parameters, as it will go through all the records, returning all the data stored in each one. It is important to note that this function was developed for proof of concept purposes, and in a production environment, its uses (if necessary) require the correct permission policies; and
- **getHistory**: The getHistory function returns the history of events associated with the container ID passed as a parameter.

The chaincode developed for the proof of concept has functions designed to validate the solution's functionalities. However, it can be updated without needing to recreate or reinstall the channel, allowing the addition of new features, modification of existing functions, among other code changes, and giving flexibility to the implemented solution. Any modification in the chaincode must be approved by the channel participants, that need to sign the chaincode proposal before its installation.

4.6.4 Collectors

The event2ledger collector is built as a container application through a script (Dockerfile), ensuring the correct installation of all necessary dependencies. Each Docker Swarm node runs a collector that has two main functions: (i) collect all service and container lifecycle events non-intrusively and (ii) send event data in transaction form to be validated by blockchain nodes. Both the Docker and the Hyperledger Fabric blockchain allow communication to be carried out through their API and also via CLI.

For the proof of concept, we chose to use CLI, both for communication with the Docker and with the blockchain nodes. Thus, the collection of events takes place

through the command *docker events –filter 'type=service' –filter 'type=container' –format '{{json .}}'*, which allows the collector to get events from services and containers in JavaScript Object Notation (JSON) format. The sending is done through the command *peer chaincode invoke*, followed by the necessary arguments, among which the access path to your private key, for transaction signature, endorsement node addresses, and event data.

Collectors running on *worker* nodes gather events related to running containers, while collectors on node *manager* collect events related to containers that make up the blockchain and running services. As a result, every service or container event generated by Docker is collected, validated, and stored in the blockchain. The Code 1 presents an excerpt of the code of a collector, configured to sign transactions with the private key of the user Admin@provider.labenv.com (Line 2), containing events of the service type (Line 7) or container (Line 17), and send them to the blockchain's endorsement nodes via the Hyperledger's CLI (lines 8 to 16 and 18 to 26).

Code 1: Collector script code snippet

```
1  # Certificate to be used when signing the transaction:
2  export CORE_PEER_MSPCONFIGPATH=${PWD}/artifacts/channel/crypto-
     config/peerOrganizations/provider.labenv.com/users/Admin@provider.labenv.com/msp
3
4  # Monitors Docker events that are of the service or container type
5  docker events –filter 'type=service' –filter 'type=container' –format '{{json .}}' | while read event
6  do
7       if [ "${type:1:-1}" = "service" ]; then
8       peer chaincode invoke -o 10.20.221.71:7050
9            –ordererTLSHostnameOverride orderer.labenv.com
10           –tls $CORE_PEER_TLS_ENABLED
11           –cafile $ORDERER_CA
12           -C $CHANNEL_NAME -n ${CC_NAME}
13           –peerAddresses 10.20.221.71:7051 –tlsRootCertFiles $PEER0_PROVIDER_CA
14           –peerAddresses 10.20.221.71:9051 –tlsRootCertFiles $PEER0_DEVELOPER_CA
15           –peerAddresses 10.20.221.71:11051 –tlsRootCertFiles $PEER0_USER_CA
16           -c '{"Args":["createEvent", "'${actorID:1:-1}'", "'${type:1:-1}'", "'${action:1:-1}'",
               "'${actorID:1:-1}'", "'${name:1:-1}'", "'${replicasnew:1:-1}'", "'${replicasold:1:-1}'",
               "'${scope:1:-1}'", "'$hour'", "'$hournano
17      else if [ "${type:1:-1}" = "container" ]; then
18      peer chaincode invoke -o 10.20.221.71:7050
19           –ordererTLSHostnameOverride orderer.labenv.com
20           –tls $CORE_PEER_TLS_ENABLED
21           –cafile $ORDERER_CA
22           -C $CHANNEL_NAME -n ${CC_NAME}
23           –peerAddresses 10.20.221.71:7051 –tlsRootCertFiles $PEER0_PROVIDER_CA
24           –peerAddresses 10.20.221.71:9051 –tlsRootCertFiles $PEER0_DEVELOPER_CA
25           –peerAddresses 10.20.221.71:11051 –tlsRootCertFiles $PEER0_USER_CA
26           -c '{"Args":["createEvent", "'${actorID:1:-1}'", "'${status:1:-1}'", "'${id:1:-1}'", "'${from:1:-1}'",
               "'${type:1:-1}'", "'${action:1:-1}'", "'${actorID:1:-1}'", "'${image:1:-1}'", "'${name:1:-1}'",
               "'${scope:1:-1}'", "'$hour'", "'$hournano'"]}'
27      fi
28      fi
29 done
```

4.7 PROOF OF CONCEPT EVALUATION

The purpose of the proof-of-concept is to show the feasibility of event2ledger, implemented in a cloud-container virtualization environment, with the Docker Swarm Orchestrator. In this scenario, the goal is to collect events of types ”service ” and ”container ” (*i.e.*, referring to the lifecycle of containers and applications) directly from the local *daemon* Docker. More precisely, ”service ” events are the result of changes in the application lifecycle state and always occur on the master node of the Docker cluster. These events result in ”container ” type events in the worker node where the application is located. In order to assess the viability of event2ledger two test scenarios were designed:

1. Generation of ”service” events through the execution of an application life cycle comprising creation, dimensioning, and finalization. For this, when created, the application has a single replica. Then the number of replicas is changed to three. When there are three replicas in execution the application is finalized. The application creation generates two events: a ”service creation ” event in the *manager* node and a ”container creation ” event in the *worker* node to which the container was assigned. When changing the number of replicas to three, a service update event is generated with the number of the previous (replicasold = 1) and new (replicasnew = 3) replicas. Consequently, ”container creation ” events are generated on the assigned worker nodes. Finally, when the application ends, a ”remove” event is generated in the manager, and ”container stop” events are generated in the workers.

2. Container events generation through CLI without orchestrator intervention. Since all nodes in the cluster have a collector that connects to the local *daemon* Docker, the event capture performed by event2ledger collectors is independent of the Orchestrator. To verify this functionality, container creation, pause, and termination events must be generated directly in one of the hosts through its CLI, and the corresponding events must be located in the ledger.

In either case, the expected result is that all events should be collected by event2ledger collectors running on the nodes that compose the Docker Swarm. For each event, a corresponding transaction must be generated, validated, and its results stored in *ledgers*. Code 2 presents some of the events collected from a test application, and Code 3 shows the transactions containing these events.

Code 2: Events collected from a test application

```
{ "Type": "service",
"Action": "create",
"Actor": {
      "ID": "ciw95xncg3in1pbte8h2kmi6a",
      "Attributes": {
            "name": "appdemo"
            }
      },
"scope": "swarm",
"time": 1625360377,
"timeNano": 1625360377017468923 }


{ "Type": "container",
"status": "create",
"id": "e025f9c0479eb55769389193d",
"from": "alpine",
"Action": "create",
"Actor": {
      "ID": "e025f9c0479eb55769389193",
      "Attributes": {
            "image": "alpine",
      "name": "appdemo.1" } },
"scope": "local",
"time": 1625360846,
"timeNano": 1625360846892301746 }
```

Code 3: Transaction generated for the events shown in Code 2

```
{ "Transaction ID": 7deciw95xncg3in
"Validation Code": 0
"Payload Proposal Hash":
  a52ee818c2dee32aa214154ee86a52ee81
"Endoser":
  {"ProviderMSP","DeveloperMSP","UserMSP"}
"Chaincode Name": eventdb
"Type": ENDORSER_TRANSACTION
"Value": {
      "Type": "service",
      "Action": "create",
      "Actor": {
            "ID": "ciw95xncg3in1pbte8h2kmi6a",
            "Attributes": {
                  "name": "appdemo"
                  }
            },
      "scope": "swarm",
      "time": 1625360377,
      "timeNano": 1625360377017468923
      "sender": "e2l_manager"
      },
"Timestamp": "2021-01-30 16:25:37.356 +0000 UTC",
"IsDelete": "false" }


{ "Transaction ID": 7de3eab67b3e9dad
"Validation Code": 0
"Payload Proposal Hash":
  5e48bb6283965c2141561b46ae8535ef9e4
"Endoser":
  {"ProviderMSP","DeveloperMSP","UserMSP"}
"Chaincode Name": eventdb
"Type": ENDORSER_TRANSACTION
"Value": {
      "status": "create",
      "EventID": "e025f9c0479eb557693",
      "from": "alpine",
      "Type": "container",
      "Action": "create",
      "ActorID": "e025f9c0479eb55769389",
      "Image": "alpine",
      "name": "appdemo.1",
      "scope": "local",
      "time": "1625360846",
      "timeNano": "1625360846892301746"
      "sender": "e2l_worker1"
      },
"Timestamp": "2021-01-30 16:25:37.356 UTC",
"IsDelete": "false" }
```

All container services and events generated on nodes where the collectors were running were collected and sent to the blockchain via transactions. A validation code ” 0 ” (Code 3, line 2) indicates that the transaction was successfully validated. During the execution of the proof-of-concept tests was possible to identify the collection of all generated events as the respective validated transactions.

4.8 ATTACK MODEL ANALYSIS

The tests and analysis described in this section were performed according to the attack model presented in Section 3.4. This model is based on the classification proposed by (HENZE et al., 2017) which subdivides attacks on monitoring solutions into four groups (*i.e.*, Retention, Modification, Insertion, and Reordering).

The purpose of event2ledger when using a blockchain as part of the solution is to apply its security mechanisms (*e.g.*, authenticity, integrity, traceability, and irreversibility) to mitigate the listed attacks. In this way, the analysis of the attack model correlates attacks and defense mechanisms, showing how each mechanism works to strengthen the security and reliability of event2ledger.

4.8.1 Retention attacks

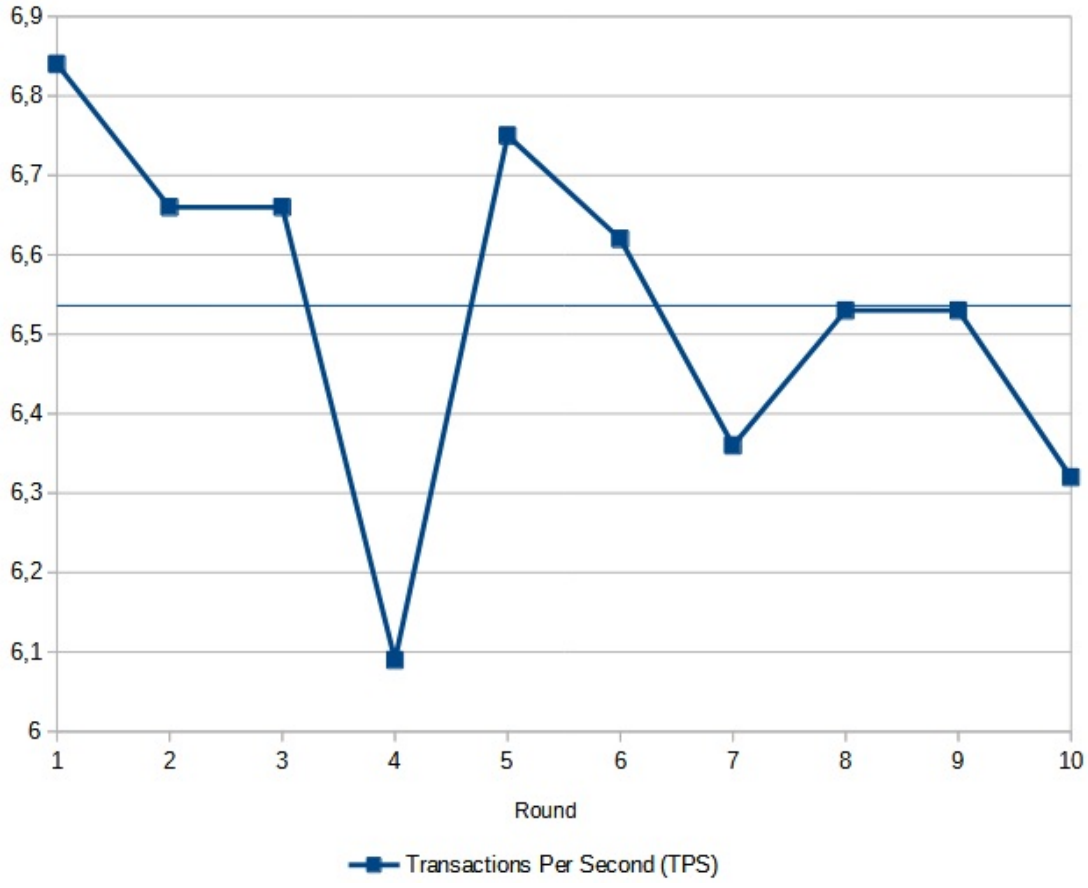
Many blockchain-based solutions have scalability issues that can lead to data retention and loss attacks. Two of the main reasons for this are the increasing amount of transactions that need to be stored in the blockchain nodes (ZHENG et al., 2018), and the reduction in the maximum *throughput* of the solution as the network grows. Both are relevant metrics to assess the scalability of a blockchain and can be influenced by characteristics such as block size, number of nodes, and endorsement policy, among others. Such issues are more critical in public blockchain-based solutions (*e.g.*, Bitcoin (NAKAMOTO, 2008)), where there is little control over the number of nodes participating in the network and the amount of data added to the blockchain. In the proposed scenario, however, the consortium blockchain adoption offers some predictability about the system throughput. This model also allows performance optimization by adjusting the block size, the time limit for its generation, and the endorsement policies.

In this scenario, considering that container mechanisms such as Docker mark the generated events with a timestamp, and the transactions also receive a timestamp during their generation, possible delays induced by an attacker are detectable through the comparison of the timestamps. This delay metric can be useful for triggering performance alarms that allow early problem detection and in-time response (*e.g.*, mark this event as late before storing so that further analysis can take this delay into account rather than being confused by it). It is also possible to define an acceptable time window within which an event is considered valid and promote consensus among interested actors. Even though this mechanism causes delays in the validation process by increasing the latency of the *ledger* update tasks, the transaction processing performance itself should not be affected, as Hyperledger Fabric adopts an “execute-order-validation” architecture, which means that *peers* can continue executing transactions while validation is still in progress. Also, since event2ledger is intended to be used as a logging system rather than a real-time event processor, it is reasonably tolerant of delays.

However, considering that the proof of concept does not address performance issues and that the implementation environments are typically dynamic and generate many events, it is relevant to analyze the performance of the solution components to assess the overall performance, possible bottlenecks, or limitations. Furthermore, as

the implementation model predicts the execution of a set of collectors in parallel, the performance analysis is also important to assess whether the blockchain *throughput* is proportional to the volume of transactions generated. In this way, the execution of the collector benchmark fed it with several events compatible with what would be observed in a production environment and then evaluating its performance. Ten rounds of sending 1000 transactions were performed, from which it was possible to calculate the average *throughput* of the collector in 6.53 Transactions Per Second (TPS). Figure 27 presents the results obtained in the test performed.

Figure 27 – Collector Benchmark Results.



Source: Author.

Similarly, blockchain performance was evaluated using the Hyperledger Caliper benchmark tool (CALIPER.DOCS, 2021), which provides performance metrics such as *throughput* and latency. Caliper was run on the *manager* node, together with the blockchain nodes, to avoid interference in the result due to possible network traffic. The tool allows the configuration of several execution parameters, as the total number of transactions to be sent, the number of executions (Rounds), and the send rate. The benchmark performed was composed of three rounds with sending 20.000 transactions at an average *sendrate* of 200 TPS. Table 9 presents the results obtained after the test execution.

Table 9 – Blockchain Benchmark Results.

Name	Success	Send Rate (TPS)	Max Latency (s)	Min Latency (s)	Avg Latency (s)	Throughput (TPS)
10000/200 Round 1	100%	190.8	0.57	0.01	0.12	190.5
10000/200 Round 2	100%	201.8	0.68	0.01	0.11	201.8
10000/200 Round 3	100%	199.7	0.79	0.01	0.17	199.6
10000/200 Round 4	100%	201.3	1.54	0.01	0.28	201.1
10000/200 Round 5	100%	197.6	0.79	0.01	0.13	197.6
10000/200 Round 6	100%	198.4	0.81	0.01	0.14	198.2
10000/200 Round 7	100%	201.5	0.91	0.01	0.15	201.4
10000/200 Round 8	100%	201.4	0.66	0.01	0.11	201.2
10000/200 Round 9	100%	201.6	1.10	0.01	0.17	201.5
10000/200 Round 10	100%	202.3	1.20	0.01	0.24	202.0

Source: Author.

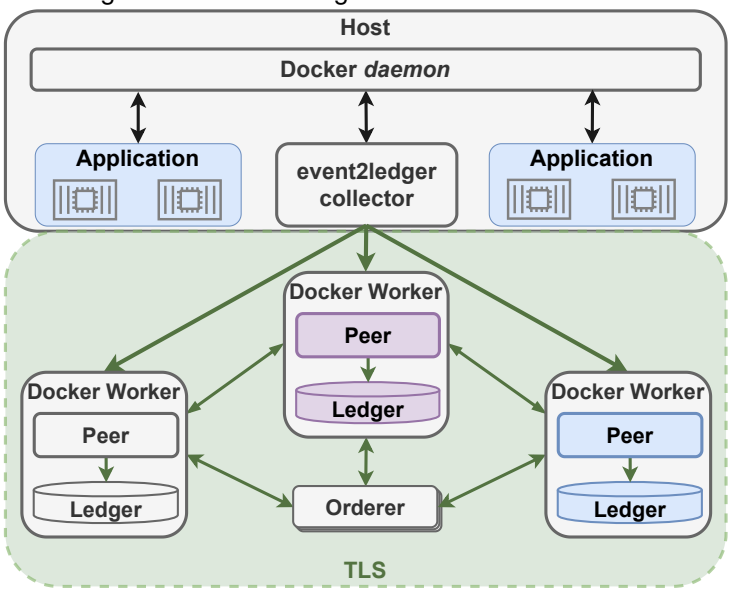
The test results show that the collector throughput is low, being a possible bottleneck for implementation in a complex production environment. It is due to the implementation model adopted, which generates transactions through a *script bash* using the Hyperledger's CLI. Despite being simple and functional, this model is only suitable for proof of concept, tests, and environments that do not demand high performance from the collector. For production environments, it is recommended to use the API or SDK when communicating with the blockchain, through which better performance can be achieved.

The results obtained with the blockchain performance test showed that the average latency of a transaction (*i.e.*, time needed to fulfill the entire ordering and validation flow) was, on average, less than 0.2 seconds, and the average number of TPS obtained was 200. During the execution of the tests, it was possible to identify that there is a limit of 2500 simultaneous transactions (*i.e.*, which are in the validation process) by the Hyperledger Fabric configurations.

4.8.2 Modification and Insertion Attacks

In general, Modify and Insert attacks are performed (1) during event collection, when an attacker takes control of one of the collectors to send modified or fake events over the network, or (2) after their storage, through tampering with the data repository. In addition to these scenarios, among the Insertion attacks are also the MitM attacks. As illustrated by Figure 28, to ensure security against attacks of this type, all data flow between event2ledger components is performed using the TLS protocol, to encrypt and protect the communication.

Figure 28 – event2ledger secure communication.



Source: Author.

In prder to strengthen the security of the collection process, each event2ledger collector signs the transactions it generates with its private key, issued by a trusted CA and verified by blockchain nodes during the endorsement phase. So, as long as the collectors' private keys remain secure, these attacks will fail. Another benefit of signing transactions is greater granularity in controlling event submission permissions. This way, in case of compromise of the private key, it is possible to revoke it at any time, limiting the effects of the attack.

The collector can also be the target of a modification attack via malicious code, as detailed in Figure 14. In this model, through tampering with the collector's code, an attacker can manipulate, create, and ignore events. The model proposed in the proof of concept is vulnerable to this type of attack. However, Section 5 presents an alternative implementation that allows checking the hash of the running binary, thus avoiding malicious code attacks.

Attacks targeting already-stored transactions, on the other hand, require a direct modification to the blockchain *ledger*. By their very nature, however, blockchain architectures mitigate these attacks by replicating validated transactions across all network nodes and linking blocks through their *hashs*. Therefore, any attempt to modify data in one of the *ledgers* creates conflicts with the copies stored by the other nodes, from which the original data can be retrieved to mitigate the attack. In the specific case of event2ledger, Hyperledger Fabric can correct any problems through a state reconciliation process, which uses the channel to synchronize the *ledgers*. This solution works by having these *ledgers* communicate continuously for blocks across the *gossip* network, repairing their state if discrepancies are identified (HYPERLEDGER.DOCS, 2021).

In order to assess the feasibility of this type of attack and verify the behavior of the solution in the event of an occurrence an example transaction was generated and the following tests were performed:

- 1 - Manipulation of one of the attributes of the example transaction through direct access to one of the *legders*; and
- 2 - Query the modified transaction, performed via API Hyperledger.

The initial purpose of this test is to identify whether manipulation in the *ledger* can impact the value stored in the blockchain. With the test execution, the query to transaction data, performed via API, resulted in the original and correct transaction values. However, when performing the same query via CLI, specifying the attacked node as the only target of the query the corrupted result was displayed. As it has mechanisms to verify the integrity of transactions, this query directed to the attacked node should be able to identify the attempt to manipulate the data and, even if it was not possible to display the correct data, an alert should be issued. Also, after handling the *ledger*, the state reconciliation engine did not retrieve the original transaction data.

4.8.3 Reordenation Attacks

In event2ledger, the chronological order of the transaction is obtained through the Raft consensus mechanism, implemented in the blockchain Hyperledger Fabric. This service comprises nodes that receive transactions from different client applications simultaneously and then collaborate to organize sets of sent transactions into a well-defined sequence, bundling them into blocks. The event's chronological order manipulation can be performed by interfering with the consensus mechanism or by manipulating the *timestamps* of the stored events. As presented in Section 4.4.2, interfering with Raft execution requires the attacker to control most of the sort nodes. Thus, in the proposed scenario, the attacker must control at least two of three nodes for the attack to be successful. In proof of concept, the ordering nodes are operated by the Provider, allowing it to unilaterally interfere with the mechanism execution. However, this problem can be easily circumvented, as each one of the actors executes one of the ordering nodes. With this, only with the joint action of two-thirds of the actors, it would be possible to manipulate the transactions in the proposed scenario.

It is important to note that the sequencing of transactions in a block is not necessarily the same as that received by the ordering service, since there may be multiple nodes receiving transactions at approximately the same time. What is important is that the ordering service places transactions in a strict order used by *peers* when validating

and confirming transactions. This strict transactions order within blocks differs the Hyperledger Fabric architecture from other blockchains where the same transaction can be into several different blocks that compete to form a chain. Instead, in Hyperledger Fabric, the blocks generated by the ordering service are final: once a transaction has been written to a block its position is considered immutable. This property prevents the creation of *forks*, ensuring that validated transactions never can be reversed or discarded (FOUNDATION, 2021). The attempt to perform a reordering attack by manipulating the *timestamps* directly in the ledger is, in this case, a modification attack for which the solution implements effective mechanisms, as presented in Section 4.8.2.

4.8.4 Analysis Summary

The event2ledger defense mechanisms were analyzed based on the attack groups listed in the Attack Model, presented in Section 3.4. Table 10 summarizes the results of the analysis for the attack groups listed.

Table 10 – Summary of Attack Model analysis.

Attack Group	Attack Type	Target	Defense mechanism	Analysis result
Retention	DoS; scalability problems;	Collectors	Event collection time window;	Permissioned blockchain configuration can handle most of scalability problems; Time window can avoid out-of-time records
Modification	Fake collector; Malicious code injection; Ledger corruption;	Collectors and Ledger nodes	Collector authentication; Ledger reconciliation; chaining data hash;	Collector authentication efficient against fake collectors; Vulnerable to malicious code; Ledger reconciliation and chaining data hash to handle Ledger manipulation;
Insertion	Fake collector; malicious code injection; MitM;	Collectors and Communication nodes	Collector authentication; TLS;	Collector authentication to avoid fake collectors; TLS to defend MitM attacks;
Reordenation	Consensus manipulation; Ledger corruption;	Ordering and Ledger nodes	Ordering through Raft mechanism; Ledger reconciliation; data hash chaining;	Raft consensus mechanism hard to manipulate (2/3 nodes); Ledger reconciliation and data hash chaining; to avoid Ledger manipulation;

Source: Author.

The main attacks listed where covered by the proposed defense mechanisms. The Retention attacks can be avoided with correct blockchain configuration and tuning, using available benchmark tools, and DoS attacks can be mitigated using collection time window. Insertion attacks can be successfully defended using collectors and nodes authentication, and TLS to encrypt communication. To Reordenation attacks, the solution implements a CFT consensus mechanism that, despite not being byzantine fault tolerant, demands that at least two of the actors acts together to manipulate the data. The modification attacks can target the collectors or the Ledger nodes, and was the only group that was not fully covered by the defense mechanisms. To avoid Ledger data manipulation, the solution uses two mechanisms: a process that perform Ledger reconciliation when any divergence is detected and data hash chaining. However, event2ledger's proof of concept assure to be vulnerable to malicious code injection attacks, as did not implement any defense mechanisms against this attack.

4.9 CHAPTER CONSIDERATIONS

The development and implementation of the proof of concept described in this work aim to prove the feasibility of the proposed solution and compliance with the established functional requirements. In this sense, a generic and compositional implementation environment similar to the production environments was specified, using the most popular open-source tools and technologies, adequate to the solution's needs. The proof of concept implementation allowed the evaluation of the solution's functionality in the test environment and to prove its compliance with the defined functional requirements.

Analyzing the attack model, it was possible to highlight the defense mechanisms proposed by event2ledger to mitigate the main vulnerabilities related to integrity and authenticity to which monitoring solutions are exposed. The mechanisms where fully effective to Retention, Insertion and Reordenation attacks. The Modification attacks, in turn, can still be performed through malicious code injection. In addition, the execution of performance tests allowed identifying possible alternatives to improve the event collector throughput. The blockchain performance test also identified a simultaneous transaction limit, defined in the Hyperledger Fabric core settings (*i.e.*, core.yaml) which should be modified or removed in case of implementation in an environment where the event generation exceeds 2500 per second.

5 COSIDERATIONS & FUTURE WORK

The event2ledger employs a blockchain-based architecture to integrate security and auditing capabilities into cloud application monitoring. In this way, it improves the confidence in collection and storage processes and reduces the possibility of conflicts between participants. Even though it was developed using Docker Swarm, it can also be integrated with other orchestrators (*e.g.*, Kubernetes), modifying the collectors to handle the formats used by events generated on the target platform. After implementing the proof of concept and analyzing the attack model it was possible to certify that the proposed functional requirements were met and confirm the feasibility of deploying it in a production environment.

The main concerns pointed by the attack model tests are related to the possibility of malicious code injection in the collectors and the ledger reconciliation process. The first threat can be mitigated with adequate security measures like access control policies or integration with authentication solutions like Secure Production Identity Framework For Everyone (SPIFFE). The ledger reconciliation process, on the other hand, consists of an important tool to avoid ledger discrepancies. However, during the proof of concept tests, it was not automatically executed after the ledger modification, as it should be. It was not clear if it was due to a misconfiguration or a problem in the blockchain mechanism, and more tests are necessary to determine the problem's origin. Table 11 presents a summary of the adopted and evaluated mechanisms against the attack model and the respective correlation with the defined functional requirements.

Table 11 – Compliance with Functional Requirements.

Functional Requirements	**event2ledger**
FR1 - Event Collection	Complied with Docker CLI
FR2 - Data Integrity	Complied with cryptographic chaining and TLS
FR3 - Authenticity	Complied with private key
FR4 - Communication security	Complied with TLS
FR5 - Chronological Order	Complied with Raft
FR6 - Distributed storage secure and auditable	Complied with consortium blockchain

In order to perform the collection of events in a non-intrusive way the solution uses Docker and Hyperledger CLI. Other possibilities includes the use of API or SDK (RF1). The integrity of the collected data is guaranteed both during communication and after storage (RF2). During the communication process between the components, the solution uses the TLS protocol which, among other functionalities, guarantees data

integrity. In addition, the cryptographic chain mechanism employed by the blockchain makes it possible to guarantee the integrity of the data even after being stored, in such a way that the alteration of any transaction has the impact of modifying and invalidating all subsequent chains compared to the original. In order to assure the authenticity of the data, the event2ledger has a private key for each instance of the collector so that only with this key it is possible to send events (RF3). The other nodes that participate in the validation, ordering, and consensus processes must also authenticate with their private key before interaction. As it is a solution that is highly dependent on communication between its components, event2ledger adopts the TLS protocol to ensure the security of all communications performed (RF4). To reach a consensus about the transaction's chronological order (RF5), event2ledger uses the Raft mechanism through the Hyperledger Fabric blockchain. Finally, the distributed, protected, and auditable storage (RF6) is obtained through a consortium blockchain composed of the environment actors.

The development of the proposed solution began with the specification, implementation, and testing of a prototype running in a single host environment. This choice aimed to focus the initial effort on developing the basic components to execute the complete collection and storage solution's flow. In the first model, each actor executed an instance of the collector, and all the collectors connected to the same Docker *daemon* to get the events. Thus, each event was collected three times and stored in three different repositories. This model proved to be ineffective in clustered environments due to scalability issues, as each additional host requires the execution of three new collectors, and each new actor in the environment needs to install a new set of collectors on the hosts. In addition, the storage model already considers redundancy and fault tolerance, making it unnecessary for all actors to participate in the collection process. Thus, the developed alternative consisted of a collection model carried out by a single collector instance in each host, which generates transactions validated by all actors, facilitating the solution's scalability and the implementation of new hosts.

The proof of concept collector uses an implementation that interacts with Docker and Hyperledger blockchain through the CLI. This model proved to be useful during the proof of concept, but in production environments, collectors that use Docker and Hyperledger API or SDK are recommended for more efficient communication and higher throughput. It was also identified during the proof of concept implementation that Hyperledger Fabric has a maximum limit of 2500 concurrent transactions. It was inserted as a mechanism to mitigate denial of service attacks on the blockchain from version 2.1.0. It is important to note that this limit does not mean that the blockchain is capable of processing 2500 TPS. Every transaction in the process (*i.e.*, which has not yet been validated and stored) should be considered within this limit. Tests were then carried

out to identify the maximum number of TPS possible to be obtained without changing the core code, where this definition is located. These tests showed that the transaction sending limit is 220 TPS, above which the solution starts to present errors resulting from the described simultaneous transaction limit. Thus, it is necessary to evaluate the implementation environment to identify whether the limit of simultaneous transactions is sufficient against the volume of events generated or if it should be adjusted. Although it is possible to remove this limitation, it is recommended to keep it as it is another defense mechanism against Retention attacks.

When considering all the processes performed by event2ledger, from event collection to storage, the collector proved to be a preferable point of attack as it can be the target of hold, modify and insert attacks. Therefore, as future work, we plan to evaluate the implementation of collectors as functions present in *chaincode*, executed through a transaction that, when executed, connects to the API Docker and generates new transactions containing the events collected. This modification increases the security of the collection process, subjecting it to the security policies defined during blockchain configuration. Furthermore, inserting the collector code as part of the *chaincode* ensures its integrity, requiring that any modification must be approved by all actors.

Another alternative to improve collector security is the integration between event2ledger and SPIFFE, a set of open-source standards to reliably identify software systems in dynamic and heterogeneous environments (SPIFFE.DOCS, 2021). Adopting SPIFFE it is possible to perform authentication and verification of the integrity of the collectors passively through a set of selectors that make up the solution and are responsible for uniquely identifying containers, processes, and microservices. SPIFFE has a selector belonging to the Unix group that makes it possible to validate the hash of the executed binaries. Thus, it is possible to implement this mechanism to avoid malicious code injection attacks, which still exist in event2ledger's proof of concept. It can also be useful to strengthen communication security, as it manages the lifecycle of certificates, facilitating communication implementation using TLS and mutual Transport Layer Security (mTLS). Thus, we also consider the integration between SPIFFE and event2ledger as future work, in addition to the feasibility study of the integration between SPIFFE and the blockchain Hyperledger Fabric.

5.1 PUBLICATIONS

Published texts:

- **MARQUES, MARCO; MIERS, CHARLES.(2021).** Impactos da composição do canal no desempenho de blockchain Hyperledger Fabric consorciada. In: Escola Regional de Alto Desempenho da Região Sul, 2021, Brasil. Anais da Escola

Regional de Alto Desempenho da Região Sul (ERAD RS 2021). pages 85-86. DOI: https://doi.org/10.5753/eradrs.2021.14781

- **MARQUES, MARCO; MIERS, CHARLES.; SIMPLICIO JR., MARCOS A.(2021).** Container Allocation and Deallocation Traceability using Docker Swarm with Consortium Hyperledger Blockchain. In Proceedings of the 11th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER, ISBN 978-989-758-510-4, pages 288-295. DOI: 10.5220/0010493302880295 Qualis A2 (Qualis CC/2016).

- **MARQUES, MARCO; MIERS, CHARLES.; KOSLOVSKI, G. P.; PILLON, M. A.(2020).** Blockchains com Hyperledger: conceitos, instalação, configuração e uso. 17a Escola Regional de Redes de Computadores. 2020. Capítulo de livro. Number of pages 26.

- **MARQUES, MARCO; MIERS, CHARLES.(2020).** Registro de ciclo de vida de contêineres usando Docker Swarm com base em blockchain Hyperledger consorciado: Desempenho e impactos. In: Escola Regional de Alto Desempenho da Região Sul, 2020, Brasil. Anais da Escola Regional de Alto Desempenho da Região Sul (ERAD RS 2020). page 157.

- **MARQUES, MARCO; MIERS, CHARLES.(2020).** Rastreabilidade de alocação e desalocação de contêineres usando Docker Swarm com base em blockchain Hyperledger consorciado. Computer on the Beach, 2020, Balneário Camboriú. Computer on the Beach, 2020.

- **MARQUES, MARCO; MIERS, CHARLES.(2020).** Rastreabilidade de alocação e desalocação de contêineres usando Docker Swarm com base em blockchain consorciado. In: XVIII Escola Regional de Redes de Computadores (ERRC 2020), 2020, JOINVILLE. XVIII Escola Regional de Redes de Computadores, 2020.

- **MARQUES, MARCO; MIERS, CHARLES.(2019).** Registro de eventos de alocação de contêineres em Blockchain. In: 17a Escola Regional de Redes de Computadores, 2019, Alegrete. Anais do 17 ERRC, 2019.

The work entitled "Rastreabilidade de alocação e desalocação de contêineres usando Docker Swarm com base em blockchain Hyperledger consorciado", published in XI Computer on the Beach, received the "Best extended summary" award of the event. The work "Container Allocation and Deallocation Traceability using Docker Swarm with Consortium Hyperledger Blockchain" was selected for publication in a special issue of Springer Nature Computer Science entitled Cloud Computing and Services Science.

BIBLIOGRAPHY

BARDSIRI, A. K.; HASHEMI, S. M. Qos metrics for cloud computing services evaluation. **International Journal of Intelligent Systems and Applications**, v. 6, p. 27–33, 11 2014.

Bernstein, D. Containers and cloud: From lxc to docker to kubernetes. **IEEE Cloud Computing**, v. 1, n. 3, p. 81–84, Sep. 2014. ISSN 2372-2568.

BOHN, R. B. et al. Nist cloud computing reference architecture. **2011 IEEE World Congress on Services**, p. 594–596, 2011.

BUCHMAN, E. **Tendermint: Byzantine fault tolerance in the age of blockchains**. Tese (Doutorado), 2016.

BUI, T. Analysis of docker security. **arXiv preprint arXiv:1501.02967**, 2015.

CADVISOR. **cAdvisor Docs**. 2020. Https://github.com/google/cadvisor/tree/master/docs.

CALIPER.DOCS. **Caliper Overview**. 2021. Disponível em: <https://hyperledger.github.io/caliper/v0.4.2/getting-started/>.

CHANDRAMOULI, R. et al. **NIST Guidance on Application Container Security**. [S.l.], 2017.

CHOO, K.-K. R. The cyber threat landscape: Challenges and future research directions. **Computers & Security**, v. 30, n. 8, p. 719–731, 2011. ISSN 0167-4048. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167404811001040>.

CIUFFOLETTI, A. Automated deployment of a microservice-based monitoring infrastructure. **Procedia Computer Science**, v. 68, p. 163 – 172, 2015. ISSN 1877-0509. 1st International Conference on Cloud Forward: From Distributed to Complete Computing. Disponível em: <http://www.sciencedirect.com/science/article/pii/S187705091503077X>.

COLLECTD. **Collectd Docs**. 2020. Https://collectd.org/documentation.shtml.

CONTI, M.; DRAGONI, N.; LESYK, V. A survey of man in the middle attacks. **IEEE Communications Surveys & Tutorials**, IEEE, v. 18, n. 3, p. 2027–2051, 2016.

DAWADI, B.; SHAKYA, S.; PAUDYAL, R. Common: The real-time container and migration monitoring as a service in the cloud. **Journal of the Institute of Engineering**, v. 12, p. 51, 02 2017.

DHILLON, V.; METCALF, D.; HOOPER, M. The hyperledger project. In: **Blockchain enabled applications**. [S.l.]: Springer, 2017. p. 139–149.

DIAMANTI. **2019 Container Adoption Benchmark Survey**. 2019. Disponível em: <https://diamanti.com/wp-content/uploads/2019/06/Diamanti_2019_Container_Survey.pdf>.

DIB, O. et al. Consortium blockchains: Overview, applications and challenges. **International Journal On Advances in Telecommunications**, v. 11, n. 1&2, 2018.

DOCKER.DOCS. **Docker Overview**. 2020. Https://docs.docker.com/engine/docker-overview/.

DOCKER.SWARM. **Docker Overview**. 2020. Https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/.

FOUNDATION, L. **Hyperledger Fabric Ordering service**. 2021. Https://hyperledger-fabric.readthedocs.io/en/release-2.2/orderer/ordering_service.html.

HENZE, M. et al. Distributed configuration, authorization and management in the cloud-based internet of things. In: IEEE. **2017 IEEE Trustcom/BigDataSE/ICESS**. [S.l.], 2017. p. 185–192.

HYPERLEDGER. **A Blockchain Platform for the Enterprise**. 2020. Https://hyperledger-fabric.readthedocs.io/en/release-2.2/index.html.

HYPERLEDGER. **Hyperledger Fabric prerequisites**. 2020. Https://hyperledger-fabric.readthedocs.io/en/release-2.2/prereqs.html.

HYPERLEDGER. **Proof of Elapsed Time (PoET) 1.0 Specification**. 2020. Https://sawtooth.hyperledger.org/docs/core/releases/1.0/architecture/poet.html.

HYPERLEDGER. **Hyperledger Fabric SDK for node.js**. 2021. Https://hyperledger.github.io/fabric-sdk-node/release-1.4/global.html#Transaction_anchor.

HYPERLEDGER.ARCHITECTURE. **Hyperledger Architecture Paper 1 Consensus**. 2017. Https://www.hyperledger.org/wp-content/uploads/2017/08/Hyperledger_Arch_WG_Paper_1_Consensus.pdf.

HYPERLEDGER.DOCS. **Hyperledger Fabric official documentation**. 2021. Disponível em: <https://hyperledger-fabric.readthedocs.io/en/release-1.4/gossip.html>.

JADEJA, Y.; MODI, K. Cloud computing - concepts, architecture and challenges. In: **2012 International Conference on Computing, Electronics and Electrical Technologies (ICCEET)**. [S.l.: s.n.], 2012. p. 877–880.

JAMSHIDI, P. e. a. The journey so far and challenges ahead. **IEEE Software**, v. 35, n. 3, p. 24–35, 2018.

JIMÉNEZ, L. L. et al. Coma: Resource monitoring of docker containers. In: **CLOSER**. [S.l.: s.n.], 2015. p. 145–154.

Jo, H.; Ha, J.; Jeong, M. Light-weight service lifecycle management for edge devices in i-iot domain. In: **2018 International Conference on Information and Communication Technology Convergence (ICTC)**. [S.l.: s.n.], 2018. p. 1380–1382.

KARMEL, A.; CHANDRAMOULI, R.; IORGA, M. Sp 800-180. nist definition of microservices, application containers and system virtual machines. National Institute of Standards & Technology, 2016.

KEELE, S. et al. **Guidelines for performing systematic literature reviews in software engineering**. [S.l.], 2007.

LAU, F. et al. Distributed denial of service attacks. In: IEEE. **Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics.'cybernetics evolving to systems, humans, organizations, and their complex interactions'(cat. no. 0**. [S.l.], 2000. v. 3, p. 2275–2280.

MELL, P. M.; GRANCE, T. Sp 800-145. the nist definition of cloud computing. National Institute of Standards & Technology, 2011.

MICROSOFT. **Ferramentas de plataforma de contêiner no Windows**. 2018. Https://docs.microsoft.com/pt-br/virtualization/windowscontainers/deploy-containers/containerd.

MIERS, C. et al. Análise de mecanismos para consenso distribuído aplicados a blockchain. SBC, 2019.

MIERS, C. C. et al. Análise do tráfego de máquinas virtuais na rede de controle de nuvens computacionais baseadas em openstack. **ERRC 2019**, 2019.

MOUAT, A. Using docker: Developing and deploying software with containers. In: . [S.l.: s.n.], 2015.

NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system. **Decentralized Business Review**, p. 21260, 2008.

NETTO, H. et al. Replicação de máquinas de estado em containers no kubernetes: uma proposta de integração. 2016.

NEWMAN, S. **Building Microservices**. 1. ed. [S.l.]: OReilly, 2015. ISBN 9781491950357.

NIELES, M.; DEMPSEY, K.; PILLITTERI, V. Y. An introduction to information security. **NIST special publication**, v. 800, p. 12, 2017.

OLIVEIRA, F. et al. **A cloud-native monitoring and analytics framework**. [S.l.], 2017.

ONGARO, D.; OUSTERHOUT, J. In search of an understandable consensus algorithm. In: **Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference**. USA: USENIX Association, 2014. (USENIX ATC'14), p. 305–320. ISBN 9781931971102.

O'CONNOR, R.; ELGER, P.; CLARKE, P. Continuous software engineering—a microservices architecture perspective. **Journal of Software: Evolution and Process**, v. 29, 04 2017.

PAHL, C. et al. Cloud container technologies: a state-of-the-art review. **IEEE Transactions on Cloud Computing**, PP, p. 1–1, 05 2017.

PAJOOH, H. H. et al. Hyperledger fabric blockchain for securing the edge internet of things. **Sensors**, v. 21, 01 2021.

PALADI, N.; MICHALAS, A.; DANG, H.-V. Towards secure cloud orchestration for multi-cloud deployments. In: **Proceedings of the 5th Workshop on CrossCloud Infrastructures & Platforms**. New York, NY, USA: Association for Computing Machinery, 2018. (CrossCloud'18). ISBN 9781450356534. Disponível em: <https://doi.org/10.1145/3195870.3195874>.

PANIZZON, G. et al. A Taxonomy of container security on computational clouds: concerns and solutions. **Revista de Informática Teórica e Aplicada**, v. 26, n. 1, p. 47–59, abr. 2019. ISSN 21752745. Disponível em: <https://seer.ufrgs.br/rita/article/view/RITA-VOL26-NR1-47>.

PORTNOY, M. **Virtualization Essentials**. Wiley, 2016. ISBN 9781119267737. Disponível em: <https://books.google.com.br/books?id=FCzVDAAAQBAJ>.

POURMAJIDI, W.; MIRANSKYY, A. Logchain: Blockchain-assisted log storage. In: . [S.l.: s.n.], 2018. p. 978–982.

RODRIGUEZ, M. A.; BUYYA, R. **Container-based Cluster Orchestration Systems: A Taxonomy and Future Directions**. 2018.

RYAN, M. Enhanced certificate transparency and end-to-end encrypted mail. In: **NDSS**. [S.l.: s.n.], 2014.

Salman, T. et al. Security services using blockchains: A state of the art survey. **IEEE Communications Surveys Tutorials**, v. 21, n. 1, p. 858–880, Firstquarter 2019. ISSN 2373-745X.

SANKAR, L. S.; SINDHU, M.; SETHUMADHAVAN, M. Survey of consensus protocols on blockchain applications. In: IEEE. **2017 4th International Conference on Advanced Computing and Communication Systems (ICACCS)**. [S.l.], 2017. p. 1–5.

SPIFFE.DOCS. **SPIFFE Overview**. 2021. Https://spiffe.io/docs/latest/spiffe-about/overview/.

SYED, H. J. et al. Cloud monitoring: A review, taxonomy, and open research issues. **Journal of Network and Computer Applications**, v. 98, p. 11 – 26, 2017. ISSN 1084-8045. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1084804517302783>.

SYED, M. H.; FERNANDEZ, E. B. The container manager pattern. In: **Proceedings of the 22nd European Conference on Pattern Languages of Programs**. New York, NY, USA: Association for Computing Machinery, 2017. (EuroPLoP '17). ISBN 9781450348485. Disponível em: <https://doi.org/10.1145/3147704.3147735>.

SYED, M. H.; FERNANDEZ, E. B. A reference architecture for the container ecosystem. In: **Proceedings of the 13th International Conference on Availability, Reliability and Security**. New York, NY, USA: Association for Computing Machinery, 2018. (ARES 2018). ISBN 9781450364485. Disponível em: <https://doi.org/10.1145/3230833.3232854>.

Tianfield, H. Cloud computing architectures. In: **2011 IEEE International Conference on Systems, Man, and Cybernetics**. [S.l.: s.n.], 2011. p. 1394–1399. ISSN 1062-922X.

TINU, N. A survey on blockchain technology- taxonomy, consensus algorithms and applications. **International Journal of Computer Sciences and Engineering O**, v. 6, may 2018.

VAQUERO, L. et al. A break in the clouds: Towards a cloud definition. **Computer Communication Review**, v. 39, p. 50–55, 01 2009.

VAUCHER, S. e. a. Sgx-aware container orchestration for heterogeneous clusters. In: **Proc. of 38th ICDCS**. [S.l.]: IEEE 38th Int. Conference Distrib. Comput. Syst. (ICDCS), 2018.

WARD, J.; BARKER, A. Observing the clouds: a survey and taxonomy of cloud monitoring. **Journal of Cloud Computing**, v. 3, 12 2014.

WU, A. **Cloud-native approach with microservices**. 2017. Https://cloud.google.com/files/Cloud-native-approach-with-microservices.pdf.

YAGA, D. et al. **NISTIR 8202 - Blockchain Technology Overview**. [S.l.], 2018.

Yang, F. et al. Delegated proof of stake with downgrade: A secure and efficient blockchain consensus algorithm with downgrade mechanism. **IEEE Access**, v. 7, p. 118541–118555, 2019.

ZHENG, Z. et al. Blockchain challenges and opportunities: A survey. **International Journal of Web and Grid Services**, Inderscience Publishers (IEL), v. 14, n. 4, p. 352–375, 2018.