On Byzantine Fault Tolerance in Multi-Master Kubernertes Clusters

Gor Mack Diouf[a], Halima Elbiaze[a,*], Wael Jaafar[b]

[a]*Université du Québec À Montréal, Montreal, Quebec, Canada*
[b]*Carleton University, Ottawa, Ontario, Canada*

Abstract

Docker container virtualization technology is being widely adopted in cloud computing environments because of its lightweight and efficiency. However, it requires adequate control and management via an orchestrator. As a result, cloud providers are adopting the open-access Kubernetes platform as the standard orchestrator of containerized applications. To ensure applications' availability in Kubernetes, the latter uses Raft protocol's replication mechanism. Despite its simplicity, Raft assumes that machines fail only when shutdown. This failure event is rarely the only reason for a machine's malfunction. Indeed, software errors or malicious attacks can cause machines to exhibit Byzantine (i.e. random) behavior and thereby corrupt the accuracy and availability of the replication protocol. In this paper, we propose a Kubernetes multi-Master Robust (KmMR) platform to overcome this limitation. KmMR is based on the adaptation and integration of the BFT-SMaRt fault-tolerant replication protocol into Kubernetes environment. Unlike Raft protocol, BFT-SMaRt is resistant to both Byzantine and non-Byzantine faults. Experimental results show that KmMR is able to guarantee the continuity of services, even when the total number of tolerated faults is exceeded. In addition, KmMR provides on average a consensus time 1000 times shorter than that achieved by the conventional platform (with Raft), in such condition. Finally, we show that KmMR generates a small additional cost in terms of resource consumption compared to the conventional platform.

*Corresponding author
Email address: elbiaze.halima@uqam.ca (Halima Elbiaze)
[1]Département d'informatique, C.P. 8888, Succursale Centre-ville Montréal (Québec) H3C 3P8 Canada

---

1. Introduction

Faced with the continuous increase in capital expenditure and operating expenditure costs of fully reliable and available Information Technology (IT) systems, companies tend towards outsourcing their IT services to specialized companies such as cloud service providers. The main advantage of this strategy is to claim an excellent service quality while paying only for the necessary and consumed resources. As for the service provider, its purpose is to meet the needs of clients by providing the required resources when demanded. A common approach is to pool (or slice) its resources to share them between several clients. In this context, many challenges emerge to provide a reliable cloud environment, e.g., quality-of-service guarantee, resources management, and service continuity.

In order to exploit efficiently the service provider's resources, the virtualization technology has been introduced [1, 2]. The latter allows the services to see the resources, e.g., servers, routers, communication links, and data storage, in a manner that is independent from the physical infrastructure/equipment, and to use these resources based on service requirements, rather than on physical granularity. In particular, servers virtualization using containers, called also containerization, has gained popularity among cloud service providers, since it addresses issues, such as the inefficient use of resources [3, 4]. Unlike full-hardware virtualization, such as VMware [1], containerization leverages virtualization at the operating system level, hence generating a lighter overhead. In such system, the resource allocation unit is the container. The latter is defined as the virtual runtime environment running atop a single operating system kernel and emulating an operating system. Several implementation platforms are available for containerization, such as LXC, OpenVZ and Docker [3, 4, 5]. Nevertheless, Docker stands out as the most interesting container-based virtualization platform as it provides the simplest lightweight and scalable way of creating and deploying containers, besides its large spectrum of use cases, including hybrid clouds [6], microservices

[7], infrastructure optimization [8] and big data [9].

In a container-based server platform, containerized applications need to be managed, i.e., a container hosting an application is dynamically deployed, run, then removed. The management of these operations in a container-based virtualization platform is called Containers Orchestration. Containers Orchestration is a complex task that requires a very light but efficient mechanism for automated deployment, scaling, and management of containers. For instance, to efficiently manage Docker containers, cloud service providers such as Google, Docker, Mesosphere, Microsoft, VMware, IBM and Oracle adopted Kubernetes as their standard platform to orchestrate containerized applications [10, 11, 12]. Kubernetes is a Google open project advocating the vision of a modular, customizable and therefore scalable orchestration platform [3]. In order to guarantee the availability and continuity of hosted applications, Kubernetes uses the Raft protocol. The latter replicates the states between the machines hosting the containers, where each state is an image of the hosted containerized applications [13, 14]. In spite of its simplicity and rapidity in the replication process, Raft protocol has major limitations when it comes to machines' failure. Indeed, Raft can only detect and correctly deal with shutdown events of machines. In other words, if a machine experiences a Byzantine (random) behavior, Raft is unable to guarantee service continuity [14, 15, 16]. Indeed, Byzantine behaviors, such as delayed, dropped, or corrupted messages, or abnormally executed processes have been widely observed in real systems, as summarized in [17].

Being conscious of the risks of software errors and malicious attacks that can push a machine into a Byzantine behavior, we propose in this paper the adaptation and integration of the BFT-SMaRt fault-tolerant replication protocol into Kubernetes. By doing so, we expect our proposed platform to resist to any type of faults while guaranteeing service continuity. To the best of our knowledge, this is the first work that proposes a Kubernetes platform tolerant to Byzantine and non-Byzantine faults.

The main contributions of this paper are summarized as follows:

1. We present an overview of Docker virtualization, Kubernetes platform, fault-tolerance within this platform and its limits.

2. We propose the Kubernetes multi-Master Robust (KmMR) platform, a platform tolerant to Byzantine and non-Byzantine faults. KmMR is based on the integration of BFT-SMaRt into Kubernetes environment.

3. We propose an efficient method to adapt and integrate the replication protocol BFT-SMaRt (written in Java) into Kubernetes (written in Golang).

4. We implement the proposed KmMR solution in an OpenStack-based cloud environment, evaluate its performances and compare it to the conventional platform, called Kubernetetes multi-Master Conventional (KmMC). Comparison is realized through experiments in non-Byzantine and Byzatine environments, where both crash and Distributed Denial-of-Service (DDoS) attacks are performed to destabilize the machines and corrupt their replication process. The obtained results confirm the effectiveness and robustness of KmMR.

The rest of the paper is organized as follows. Section II describes Docker containerization technology. In section III, the Docker containers orchestration platform Kubernetes is explained. Whereas, section IV discusses fault-tolerance in Kubernetes. Section V presents our KmMR platform. Experimental evaluation and results are discussed in Section VI. Finally, section VII closes the paper.

2. Background

In this section, we present an overview of Docker containers and its orchestration mechanism, supported by Kubernetes.

2.1. Docker Containers

Container virtualization, also known as containerization, relies directly on kernel functionalities to create isolated virtual environments, as illustrated in Fig. 1. These virtual environments are named containers, while the features provided by the operating system kernel are called *namespaces* and *cgroups* [19]. The *namespaces* control and limit the amount of resources used for a process, while the *cgroups* manage the resources of a process group. Hence, a container provides the resources needed to run
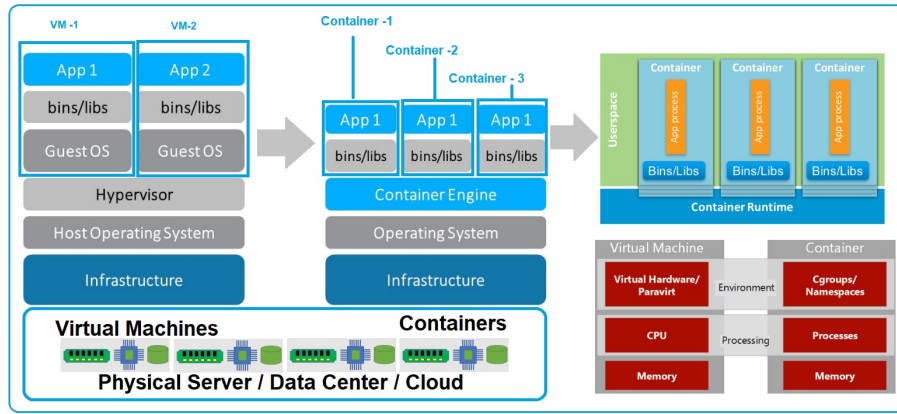
Figure 1: From virtual machines to containers [18]

applications as if they were the only processes running in the host machine's operating system.

Even though several containerization platforms have been proposed, such as LXC, OpenVZ and Docker [3, 4, 5], only Docker sparked interest and popularity among the research and professional communities thanks to its operational simplicity and flexibility. Indeed, traditional virtualization uses a Hypervisor to create virtual machines, deploy guest operating systems on them, and host the applications [20]. Whereas, Docker containerization requires only the installation of the Docker container engine on the operating system's kernel of the host machine, allowing the creation of containers that host the applications. Nevertheless, both are autonomous systems that use a higher system, i.e., the one of the host machine, to perform their tasks. The difference is that virtual machines must contain a whole guest operating system, while the containers use directly the one of the host machine. Fig. 1 illustrates an architectural comparison of traditional virtualization and containerization.

Docker is a complex but very intuitive ecosystem for container development. It is mainly composed of six elements, as shown in Fig. 2:

1. *Docker Client:* It is the command line interface tool used to configure and interact with Docker. For any Docker command instruction (e.g., $docker\ run$), the client sends the command to Docker daemon ($dockerd$) that carries it out.
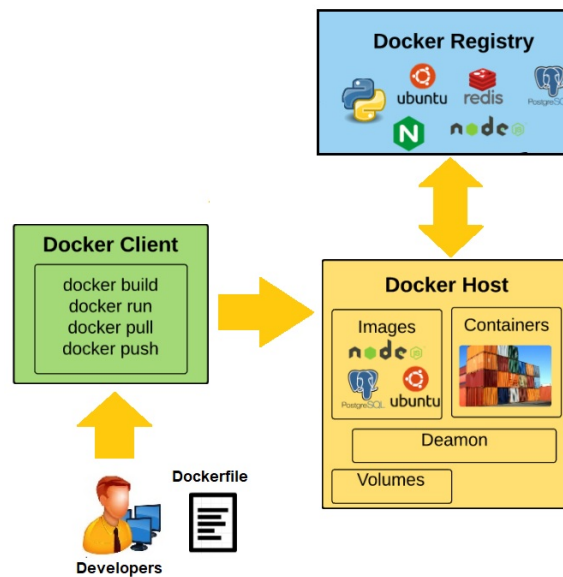
Figure 2: Components of Docker [18]

2. *Docker Daemon:* It is the Docker server that listens to the application programming interface requests and manages Docker objects, such as images, containers, networks, and volumes, etc.

3. *Docker Images:* An image is a read-only template/snapshot, pulled or pushed from a public or a private repository, in order to create a Docker container. This is the building block of Docker. It is lightweight, small, and fast compared to those of traditional virtual machines.

4. *Docker file:* It is used to build Docker images.

5. *Docker Containers:* Basically, a Docker container is a user space of the operating system. It is composed of a set of processes isolated from the rest of the system, and running from an image that provides necessary files to support the processes of the hosted application.

6. *Docker registries:* Registries are the central repository and distribution component of Docker images.

7. *Docker Engine:* It combines the Docker daemon, application programming interface and the command line interface tools.

By its simplicity and small number of components, the Docker architecture provides interesting advantages [21, 22, 23]:

1. *Deployment rapidity:* Docker achieves fast operations, such as communication, and container building, testing and deployment.

2. *Applications Portability:* Containerized applications are easily portable, as they can be moved around as a single unit, without affecting their response performances or the containers.

3. *Fast service delivery:* Docker containers format is standardized, such that programmers and administrators tasks do not interfere when deploying them. Indeed, Docker provides a reliable, consistent, and enhanced environment that achieves predictable outputs when codes are moved between development, test and deployment platforms.

4. *Density:* Docker uses the available resources more efficiently compared to virtual machines, since it does not rely on a Hypervisor. It is able to run densely several containers on the same single host, hence optimally using the resources and increasing its performance, compared to virtual machines.

5. *Scalability:* Docker can be deployed in several physical servers, data servers, and cloud platforms without any restriction. Containers can be easily moved from a cloud environment to a local host and vis-versa, at a fast pace. Deployment adjustments can be easily realized according to needs.

In Table 1, we summarize the characteristics of virtual machines and Docker containers. Accordingly, the Docker container can be created and removed almost in real time and thus introduces a negligible task overload with respect to the host machine's resources use [22, 24, 25]. Compared to virtual machines, Docker containers are advantageous in terms of network management, boot speed, deployment/migration flexibility, and resources use, e.g. RAM, storage, etc. [22]. However, they suffer from the weak isolation of the host machine. Indeed, if a Docker container is compromised, then an attacker can get full access to the operating system of the host machine [26, 27]. Consequently, there is an urgent need for a robust and secured environment for Docker containers. Moreover, Docker is unable on its own to deploy containers on distributed

Table 1: Characteristics of Virtualization Technologies

Parameters	Virtual Machines	Docker Containers
Operating System	Every virtual machine virtualizes the host material and loads its own guest operating system	No container emulates host material. Host operating system is used.
Communication	Through Ethernet peripherals	Through Inter-Process Communication standard mechanisms, e.g., sockets, pipes, shared memory, etc.
Resources Usage (CPU and RAM)	High	Quasi-native
Startup time	Few minutes	Few seconds
Storage	High requirement for guest operating system and associated software installation and execution	Low since host operating system is used
Isolation	Libraries and files' sharing among virtual machines is impossible	Libraries and files can be seamlessly mounted and shared
Security	Depends on the Hypervisor's configuration	Requires access control

machines and ensure their interaction [26]. In this matter, an orchestration mechanism is needed to manage Docker containers in distributed systems.

2.2. Containers Orchestration

Handling a few Docker containers on one machine is an easy task. However, when it comes to moving these containers into production on a set of distributed hosts, many questions arise. Indeed, driven by providing availability, scaling, and networking, an integration and management tool is required not only to ensure initial containers deployment, but to also manage multiple and dynamic containers as one entity. Clearly, handling everything manually is not conceivable because it would be very difficult to ensure the viability, maintenance and sustainability of the system. Thus, the process of deploying multiple containers can be optimized through automation, especially in large scale systems. This type of automation is referred to as orchestration and includes features like work nodes' location determination, load balancing, inter-container communication, service discovery, status updates, containers migrations, scaling up, and tolerance to malfunctions.

Several orchestrators have been proposed and implemented to manage Docker container-based platforms. Examples include Fleet [28], Mesos [29], Swarm [30] and Kubernetes [31]. In the remainder of this paper, we are interested in Kubernetes only. The latter is a stable and free solution that can automate the deployment, migration, monitoring, networking, scalability, and availability of applications hosted in container-based server platforms [4, 11].

3. Kubernetes: An Open-Access Orchestrator of Docker Containers

Kubernetes, abbreviated K8s, is a project initiated by Google in 2014 when it saw the advantages of Docker containers over traditional virtualization. The Kubernetes Orchestrator automates the deployment and management of large-scale containerized applications, such as applications' microservices generation [7], cloud services to store, access, edit and share video content [32], and mission critical services as telecommunications and energy delivery services [33, 34]. Its platform runs and coordinates containers on sets of physical and/or virtual machines. Kubernetes is designed to fully
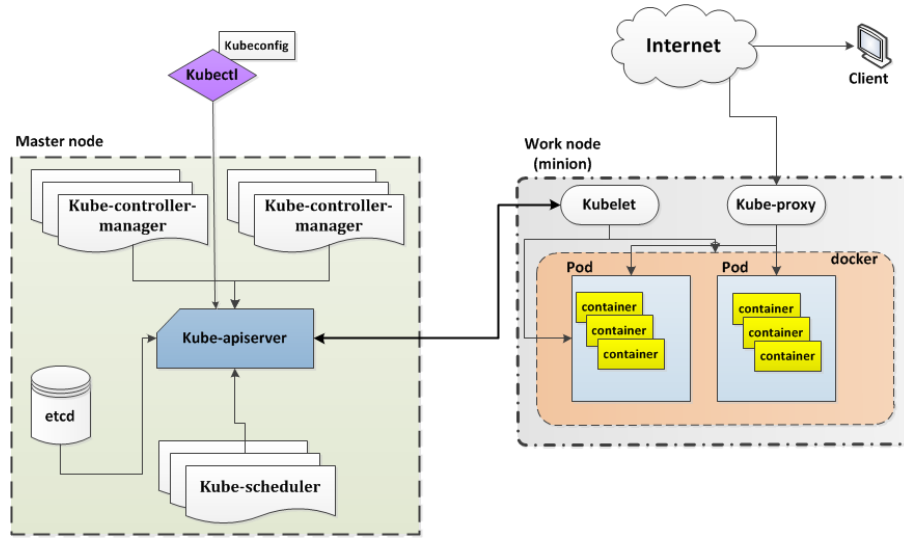
Figure 3: Architecture of Kubernetes (ex: one master node and one work node)

manage the life cycle of containerized applications, using predictability, extensibility, and high availability methods, as detailed in [35, 36].

3.1. Kubernetes Architecture

Kubernetes architecture is based on the master/slave model [37]. It consists of a cluster of one master node and several work nodes, called $minions$, as shown in Fig. 3. Their roles are given as follows:

Kubernetes master: This node is responsible of the overall management and availability of the Kubernetes cluster. Its components, i.e. the Application Programming Interface (API) server, controller and scheduler, support the interaction, monitoring and scheduling tasks within the cluster. The API server provides the interface to the shared state of the cluster through which the other components, e.g. work nodes, interact. The controller monitors the shared state of the cluster through the API server and makes decisions to bring the cluster back from an unstable state to a stable one. The scheduler manages the cluster load. It takes into account individual and collective resource requirements, quality-of-service requirements, hardware/software constraints, policies, etc. The Kubernetes cluster data is stored in a database, e.g. etcd [38], whereas cluster

administration is at the master level via the K8s command-line interface $kubectl$. The latter stores its configuration and authentication information to access the API server in the $kubeconfig$ file.

Kubernetes minions: Containerized applications run on these nodes. On one hand, the client nodes communicate with the work node via their $kubelet$ through the master node. The $kubelet$ receives commands from the master node and executes them through its Docker engine. It also reports the state of the work node to the API server. On the other hand, the *kube-proxy* runs on each work node to manage clients' access to deployed services. Each service is compiled into one or many *Pods*. A *Pod* is a logical set of one or several containers. This is the smallest unit that can be programmed as a deployment in Kubernetes. Containers in the same *Pod* share resources such as storage capacity, IP address, etc.

3.2. Pods Instantiation

In Kubernetes, the placement of *Pods* is realized following a specific strategy. In fact, considering a Kubernetes cluster consisting of a master node and a finite set of minions $\mathcal{M} = \{M_1, M_2, ..., M_n\}$, a pod $P(t, m, p, v)$ asking for t CPU cycles, m RAM, a specific communication port p and a v storage capacity, needs to be deployed within the cluster. To select the $minion$ on which the pod will be instantiated, the K8s master node proceeds in two steps: 1) it filters the *minions*. Then, 2) it ranks the remaining *minions* to determine the best one suited for the pod. These two steps are detailed as follows:

Filtering: In this operation, nodes without required resources (t, m, p, v) are removed. Kubernetes uses multiple predicates to perform filtering, including:

- *PodFitsResources*: does the node have enough resources (CPU and RAM) to accommodate the pod?

- *PodFitsHostPorts*: is the node able to run the pod via the p port without conflicts?

- *NoVolumeZoneConflict*: does the node have the amount of v storage that the pod requests?

- *MatchNodeSelector*: does the node match the parameters of the selector query defined in the pod description?

These predicates can be combined to set up sophisticated filters.

Ranking: After filtering, Kubernetes uses priority functions to determine the best $minion$ among the nodes able to host the pod. A priority function assigns a score between 0 and 10 where 0 is the least preferred and 10 is the most preferred node. Each priority function is weighted by a positive number and the final score is the sum of the weighted scores. The main priority functions that can be activated in Kubernetes are:

- *BalancedResourceAllocation*: it aims at balancing the $minions$ charge. Indeed, it places the pod in a node in a way that the resource utilization rate (CPU and RAM) is balanced among the minions.

- *LeastRequestedPriority*: it favors the node that has most resources available.

- *CalculateSpreadPriority*: it minimizes the number of pods belonging to the same service on the same node.

- *CalculateAntiAffinityPriority*: it minimizes the number of pods belonging to the same service on nodes sharing a particular attribute or label.

- *CalculateNodeLabelPriority*: it favors nodes with a specific label.

Once the final scores of all nodes are calculated, the $minion$ having the highest score is selecte to instantiate the pod. If there is more than one $minion$ that has the highest score, the master node selects one of them randomly.

4. Fault Tolerance in Kubernetes

In this section, we explain the fault tolerance mechanism in Kubernetes. We start by a brief description of faults. Next, we present the associated consensus problem. Finally, the built-in fault tolerance protocol "Raft" is detailed.

4.1. Background

The robustness of a system refers to its ability to continue functioning when part of the system fails [39]. A system fails when the outputs are no longer conform to the original specification. The occurrence of a failure can be: 1) transient, i.e. appears, disappears and never occur again, 2) intermittent, i.e. reproducible in a given context and 3) persistent, i.e. appears until repair. A non-faulty (non-failing) node or process is called correct when it follows its specifications. Whereas, a faulty node/process may stop or exhibits a random behavior. In general, failures/faults may be caused by software defects, malicious attacks, or human-machine interaction errors. In distributed systems orchestrated by Kubernetes, faults may occur at the master node or minions. They can be classified into two categories:

1. *Fail-stop faults*: They are characterized by the complete activity's stop (or crash) of a node. This state is perceived by others as the absence of expected messages until the eventual application's termination. A system that is able to detect only these faults considers that a node/process can be in one of two states, either it works and gives the correct result, or it does nothing.

2. *Byzantine faults*: Byzantine faults are characterized by any behavior deviating from the node/process's specifications and producing non-conform results [40]. We distinguish between natural Byzantine faults, such as undetected physical errors on messages' transmissions, memory and instructions, and malicious Byzantine faults, designed to defeat the system, such as viruses, worms and sabotage instructions.

In large and/or uncontrolled systems, the risk of faults is high and shall be mitigated to ensure service continuity. One way to realize it is to use the State Machine Replication (SMR) mechanism [41]. The latter consists of using multiple copies of a system, implemented as a state machine, to tolerate faults and keep the system's availability. Each copy of the system, called a replica, is placed on a different node [42]. SMR allows a set of nodes to execute the same instruction sequences on each request sent by a client. There are two approaches to execute requests: 1) active replication, where all nodes execute requests, update their state machines, and respond to clients. And 2) passive

replication, where only one node, called *leader*, executes the requests and forwards state machine changes to other nodes, then responds to clients.

To avoid inconsistency in replication, nodes/replicas need to be sure that their state machines are identical before responding to clients. The following section describes this state machine replication problem, called the *Consensus* problem.

4.2. Consensus Problem

The *Consensus* is a fundamental condition in fault-tolerant distributed systems. It consists of tuning replicas' values to the same one, proposed by one of the nodes. The *Consensus* problem can be formulated as follows: We assume a system composed of a set $\mathcal{N} = \{N_1, N_2, \ldots, N_n\}$ of n replicated nodes, and that at most only f nodes can fail, where $f \leq n - 1$. Let $\mathcal{N}' \subseteq \mathcal{N}$ be a subset of $m \leq n$ nodes. The consensus problem consists of finding a protocol that allows the following:

1. Any node $\in \mathcal{N}'$ can propose a replica's value to the other nodes.

2. When all nodes agree on the same value, a consensus is achieved.

Without loss of generality, protocols that satisfy these conditions, possess four properties [43]:

1. *Termination*: Each correct node eventually decides a value.

2. *Validity*: The decided value has been proposed by one or many other nodes.

3. *Integrity*: The decision is unique and final.

4. *Agreement*: Two correct nodes cannot decide different values.

According to [42], any protocol that verifies the following safety and liveness conditions has the previous four properties:

1. *Safety*: All the correct replicas execute the requests they receive in the same order.

2. *Liveness*: Each request is correctly executed by correct nodes.

Such a protocol is commonly referred as consensus/replication protocol. Its decisions are based on exchanged messages between all or a part of the nodes in the system.

Indeed, a consensus is achieved if the quorom, defined as the minimum number of correct nodes required to build the consensus, participate in the consensus process. The quorum depends on the size of the system and the maximum number of tolerated faults. Two fault-tolerant classes of replication protocols exist. In the first, called *Non-Byzantine*, nodes fail only when they stop functioning. For n nodes, at most $f = \frac{n-1}{2}$ crash faults can be tolerated. Examples of non-Byzantine protocols include Raft [14], Paxos [44], and Zab [45]. In the second class, called *Byzantine*, any type of failures can be tolerated. However, they typically tolerate only $f' = \frac{n-1}{3}$ faults [46]. As Byzantine protocols examples, we can cite Practical Byzantine Fault-Tolerance (PBFT) [47], Efficient Byzantine Fault-Tolerance (EBFT) [48], UpRight [49], Prime[50], and Byzantine Fault-Tolerance State Machine Replication (BFT-SMaRt) [15, 51].

4.3. Built-in Fault Tolerance in Kubernetes: Raft Protocol

Raft is the replication protocol built into Kubernetes [14, 52]. Basically, it ensures that the replicas maintain identical state machines, while tolerating only crash faults. It is based on passive replication, where a node may be *leader*, *follower* or *candidate*, as illustrated in Fig. 4:

- *Leader*: In a cluster, a single active node directs the communication, by receiving requests, processing them, forwarding state machine changes to other nodes, and responding to clients.

- *Follower*: When a *leader* is active, all other nodes are set as *followers*. They wait for the changes sent by the *leader* to update their state machines.

- *Candidate*: When the *leader* breaks down, the *followers* become *candidates* and trigger votes to elect a new *leader*.

The *mandate* of a *leader* lasts from its election until its breakdown. In order to organize elections, Raft assigns an index to each mandate. These indexes are called *terms*. Any leader or candidate node includes the *term* index in its messages. Whereas, a *follower* needs to wait for a random time, typically between 150 and 300 ms, before transiting into *candidate*. An active *leader* periodically sends heartbeat messages
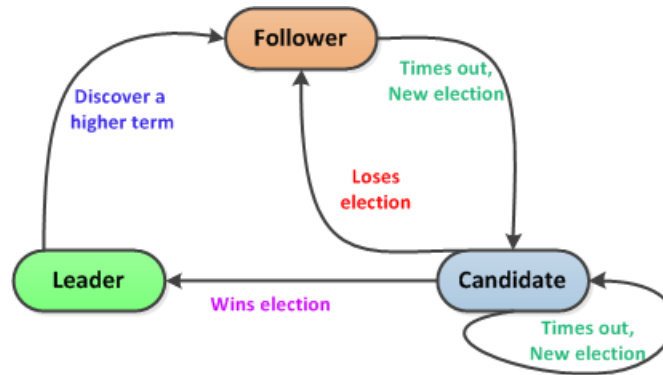
Figure 4: Raft Protocol's Election Process

($AppendEntriesMessage$) to all nodes in the cluster. Any node receiving this message resets its wait time to a random value. Otherwise, at the expiration of its wait timer, the *follower* changes status to *candidate* and triggers a new election. The *candidate* proceeds as follows: 1) Increments its current term number, 2) votes for itself, and 3) sends vote request ($RequestVoteMessages$) to all other nodes. The latter vote for the request containing a *term* index greater than theirs, update their term index and return to the *follower* status. Once a candidate receives the votes of the majority, defined as $\lceil f + 1 \rceil$ votes, it becomes the new *leader*. However, if no candidate obtains the majority of votes, e.g. in a tie situation, no leader is elected in this term, and a new term will be triggered by the node that sees its timer expiring first. The requirement for a majority of votes ensures that a single *leader* is elected in a *term* (*Safety* condition), while the wait time of *followers* guarantees that a leader will eventually be elected (*Liveliness* condition).

To run in Kubernetes environment, some changes have been made to Raft protocol:

1. Unlike the conventional Raft, where requests to *followers* are redirected to the *leader*, Raft is converted to active replication to be conform to the load balancing property of Kubernetes [52].

2. Raft is re-implemented in Golang, the same programming language used to develop Kubernetes and Docker containers.

Besides Raft, another non-Byzantine replication protocol, called $DORADO$ was pro-

posed for Kuberenetes [53]. This protocol is similar to Raft, but requires sharing the master node's memory to all instanciated containers in work nodes, in order to store their state machines. This approach allows to achieve shorter consensus times than Raft, but aggravates the containers' isolation issue.

Despite their simplicity, Raft and $DORADO$ are particularly powerless against Byzantine behaviors [54]. Indeed, a failing node may not stop, and adopts continually a Byzantine (random) behavior, e.g. not following the protocol, corrupting its local state, or producing incorrect or inconsistent outputs [42]. To mitigate this problem, we propose in the next section a novel Kubernetes platform, where both non-Byzantine and Byzantine faults can be tolerated, while ensuring service continuity.

5. KmMR: A K8s multi-Master Robust Platform

Kubernetes allows to deploy and orchestrate groups of containers with a single master node. The latter replicates the containers on different work nodes to provide service continuity. However, if the master node fails, containers are no longer available and all management data is lost. To avoid such case, the deployment of multi-master clusters, where several master nodes cooperate, becomes necessary. However, duplicating master nodes only does not provide complete fault tolerance [55]. In fact, this mechanism must be associated with a replication protocol to ensure consistency between the master nodes states, i.e., update operations to a replicated data item within the nodes should reach and be executed at some time, in all master nodes, and in the same chronological order [42, 56]. Such multi-master systems are important for critical applications, e.g., telecommunication and energy services, where the continuous availability of services is required 24 hours a day, and 7 days a week.

In this section, we propose to create a resistant Kubernetes multi-master platform to all kinds of faults, in order to guarantee service continuity. We consider a Kubernetes cluster consisting of n replicated K8s master nodes and c work nodes. Work nodes process clients' service requests and send their reports (requests) to the master nodes, as shown in Fig. 5. We assume that communications between nodes may experience important delays, thus causing communication failures.
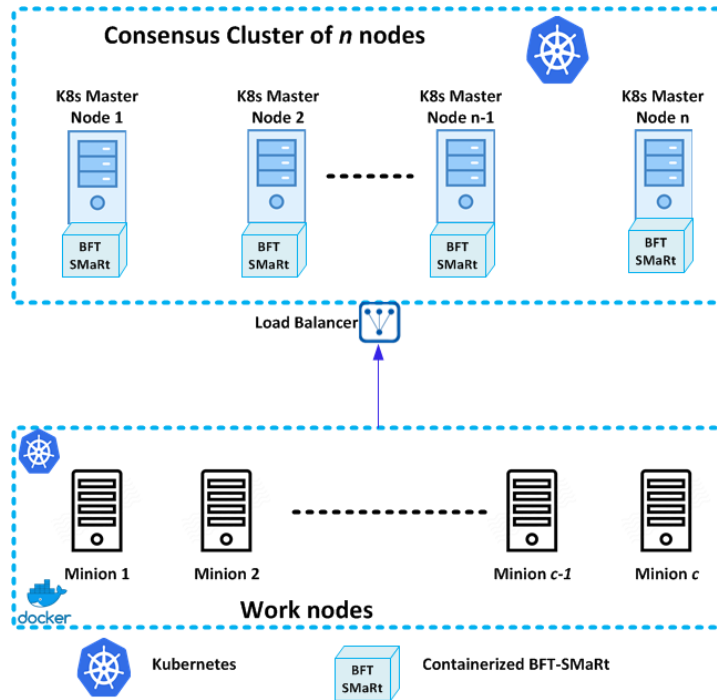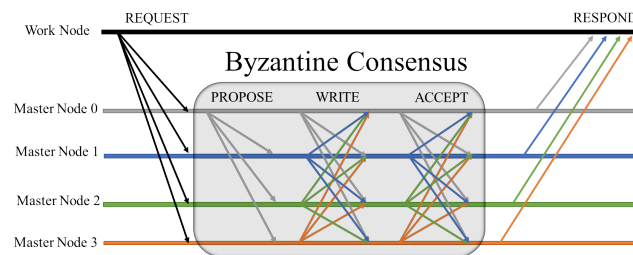
Figure 5: System Model

Figure 6: Consensus Process by BFT-SMaRt

5.1. BFT-SMaRt: Replication Protocol for KmMR

Among the known Byzantine protocols, only PBFT [47], UpRight [49] and BFT-SMaRt [15] implement a Byzantine fault-tolerant replication system. The choice of BFT-SMaRt is motivated by the following:

- BFT-SMaRt is very well suited for modern hardware, e.g. multi-core systems, unlike other protocols such as PBFT [15].

- BFT-SMaRt outperforms other protocols, e.g. UpRight, in terms of consensus time, defined as the required time to process a client's request [15].

- BFT-SMaRt guarantees a high accuracy in replicated data, when a Byzantine faulty behavior is exhibited within the system [15].

- BFT-SMaRt is a modular, extensible and robust library. It is able to provide an adaptable library that sets-up reliable services [57].

- Unlike other Byzantine protocols, BFT-SMaRt supports reconfiguration of the replica sets, e.g., addition and removal of nodes [58].

- BFT-SMaRt provides efficient and transparent support for critical and sustainable services [59].

In BFT-SMaRt, a consensus is established according to the following steps, as illustrated in Fig. 6. First, a work node broadcasts its request to master nodes, who trigger the execution of the consensus protocol. Each instance of the consensus begins with the *leader* master node proposing to other nodes a batch of requests in the *PROPOSE* message. Master nodes validate the authenticity of the *PROPOSE* message and its content. If valid, they register the proposed batch and broadcast *WRITE* messages with cryptographic hashes of the proposed batch, to all other nodes. If a master node receives $\lceil \frac{n+f'+1}{2} \rceil$ *WRITE* messages with the same hash, where $\lceil . \rceil$ is the ceiling function, it sends an *ACCEPT* message to all other nodes. This message contains its decision batch for the consensus instance. If the *leader* master node is not correct, a new election must be triggered, and all nodes need to converge to the same execution by consensus. The election procedure is described in detail in [60].
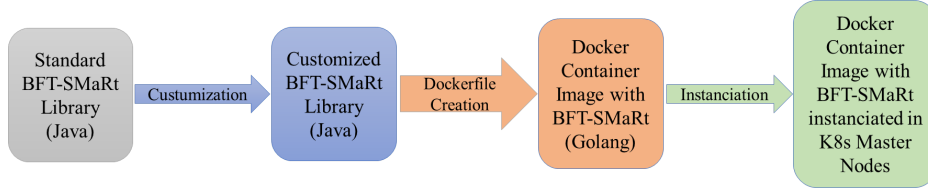
Figure 7: Integration Methodology of BFT-SMaRt into Kubernetes

5.2. Proposed Integration Methodology of BFT-SMaRt into K8s

The BFT-SMaRt protocol is implemented in Java, an object-oriented programming language, while Kubernetes and the Docker engine are written in Golang, a service-oriented programming language [61]. In order to integrate BFT-SMaRt into Kubernetes, two options can be considered:

1. Rewrite all BFT-SMaRt library's source code in Golang.
2. Wrap the BFT-SMaRt library in a Docker container.

Unlike Raft, with a source code less than 3000 lines and easily rewrited in Golang, BFT-SMaRt source code is larger and more complex, with approximately 100 files and a total of 13500 lines of Java code. Consequently, the second option is more likely to be realizable. This choice is supported by the advantages offered by Docker. Indeed, Docker containers run fast and their introduced overhead is negligible [24, 22]. The proposed procedure to integrate BFT-SMaRt into Kubernetes is illustrated in Fig. 7. First, we recover the library BFT-SMaRt and all its dependencies from Github [57]. Then, we customize it by setting the parameters of the master nodes. Next, we create our Docker file *Dockerfile*, as detailed in Fig. 8. Afterwards, we execute *Dockerfile* to produce the BFT-SMaRt containerized image. Finlly, we instantiate in each K8s master node the Docker image with its information.

6. Experimental Evaluation

6.1. Simulation Settings

We implemented the KmMR solution in an OpenStack cloud environment provided by Ericsson Canada [62]. The available resources are as follows: 50 GB of RAM and 20 virtual processors (VCPU), usable on a maximum of 10 machines.

```
# This will build the BFT-SMaRT image
FROM  openjdk:8-jre-alpine

# Copy customized library to image
COPY  lib/                    usr/lib/
COPY  config/          usr/config/
COPY  bin/BFT-SMART.jar usr/bin/BFT-SMART.jar

# Expose container's port
EXPOSE 1000

# Mount volume for container's port
VOLUME /volume/data/

# Run library's scripts
CMD ["runscripts/smartrm.sh"]
```

Figure 8: Dockerfile to Create the BFT-SMaRt Container

The experiment is carried out on clusters composed of several Kubernetes master nodes ($n = 5$ and $n = 7$), connected to each other via the OpenStack GigabitEthernet network and accessible from the Internet. Each node is a virtual machine equipped with the Ubuntu server 18.04 TLS 64-bit OS, a dual-core i7 CPUs (VCPU) clocked at 2.4 GHz, 4 GB of RAM and 20 GB storage capacity. The Docker engine 18.05.0-ce is installed on Kubernetes nodes for container instantiation needs. We deployed Kubernetes 1.11.0 to orchestrate the Docker containers. The master Kubernetes role *kubeadm* has been enabled on all master nodes (multi-master configuration). The remaining machines are used to act as work nodes and DDoS attackers. BFT-SMaRt has been containerized and integrated into the master nodes to provide coordination and consensus. Work nodes send their requests in closed loop, i.e. they wait for the response of a request before sending a new one, as defined in [63].

In the cluster, we initialize the replication protocol on master nodes. Then, two work nodes broadcast their requests. Upon request reception, master nodes exchange messages to build the consensus. To measure the performance of KmMR, we used the micro-benchmark $0/0$ where both request and response messages are empty [47].

DDoS attacks are used to model Byzantine behaviours, using the *Hping3* command

[64, 65]. Indeed, we inject DDoS-based "CPU Load" and "Network Flooding" Byzantine faults as follows [66, 67]. "CPU Load" fault is triggered by increasing the number of users continuously sending requests to a master node, while "Network Flooding" can be initiated by some master nodes towards others. We assume that attacking machines target simultaneously a single master node. Each attacker sends successively and continuously requests of size 65495 bytes in open loop, i.e. without waiting for responses, through the command *Hping3 -f IP address of targeted master node -d 65495*.

We evaluate the performance of our solution and compare it to the Kubernetes multi-Master Conventional (KmMC) platform, where non-Byzantine replication protocol Raft is used. Two scenarios are considered for our experiments:

- *Scenario 1*: In this scenario, we consider a Kubernetes platform where, initially, the number of (crash) faults in the cluster is lower than the maximum number of faults tolerated by the replication protocol in place. This corresponds to $f < \frac{n-1}{2}$ and $f' < \frac{n-1}{3}$ for KmMC and KmMR respectively. Then, we perform a DDoS attack on one master node, and evaluate the consensus times for each platform.

- *Scenario 2*: Unlike *Scenario 1*, the initial number of (crash) faults is set to be the maximum that can be tolerated by the used replication protocol. Then, DDoS attacks are performed on one master node. In this scenario, we evaluate established consensus times as well as resources consumption by the DDoS victim (CPU, RAM, and available communication Bandwidth). Resources are measured using commands *IPerf3* for Bandwidth, and *top* for CPU and RAM [68, 69].

6.2. Results and Discussions

Considering *Scenario 1*, we present in Table 2 the achieved consensus times versus DDoS attack rate of KmMC and KmMR, for a cluster of 5 and 7 master nodes respectively. For both platforms, consensus times increase slightly and proportionally to DDoS attack rates. Indeed, even with the additional Byzantine fault, f and f' respect the maximum number of tolerated faults[2]. Hence, platforms' operation continue

---

[2]Notice that KmMC sees the DDoS attack as a crash event in this case.

Table 2: Consensus Times (μsec) versus DDOS Attack Rate (*Gbps*) (*Scenario 1*)

DDoS attack rate	KmMC		KmMR	
	5 K8s Master Nodes	7 K8s Master Nodes	5 K8s Master Nodes	7 K8s Master Nodes
0	1701.91	2048.25	2746.45	3161.83
2	2004.38	2132.93	2940.87	3179.45
4	2178.72	2471.39	3362.42	4521.79
4.5	2201.37	2501.73	3525.17	4632.38
5	2287.65	2623.87	3612.93	4729.98
5.5	2304.12	2702.99	3867.32	4970.93
6	2331.12	2732.25	4053.53	4970.93

without significant degradation. However, KmMC realizes shorter consensus times than KmMR. This is expected, since the replication protocol Raft is designed with few consensus message exchanges between master nodes, compared to BFT-SMaRt. Finally, we conclude that it is recommended to select the KmMC platform if the risk of exceeding the maximum number of faults, dictated by Raft, is very low.

For *Scenario 2*, we present in Fig. 9 the consensus time versus DDoS attack rate, for a cluster of 5 master nodes. The results show that the consensus time increases with DDoS attack rate. When the attack rate is below 4.25 Gbps, KmMC provides a slightly better performance than KmMR. Indeed, in this case, the DDoS victim resists to the attack thanks to its sufficient resources. However, for an attack rate above 4.25 Gbps, KmMC deteriorates rapidly and significantly. This is mainly due to the vulnerability of Raft replication protocol in front of Byzantine faults. Indeed, the DDoS victim would behave improperly, e.g. not responding to other nodes in a timely manner. Thus, from this moment, Raft triggers changes in the cluster's leadership since it is no longer able to reach a consensus with its current *leader*. This triggering considerably slows down consensus in the KmMC platform. Meanwhile, KmMR resists to all DDoS attacks, and is able to achieve consensus time 1000 times better than KmMC in average.
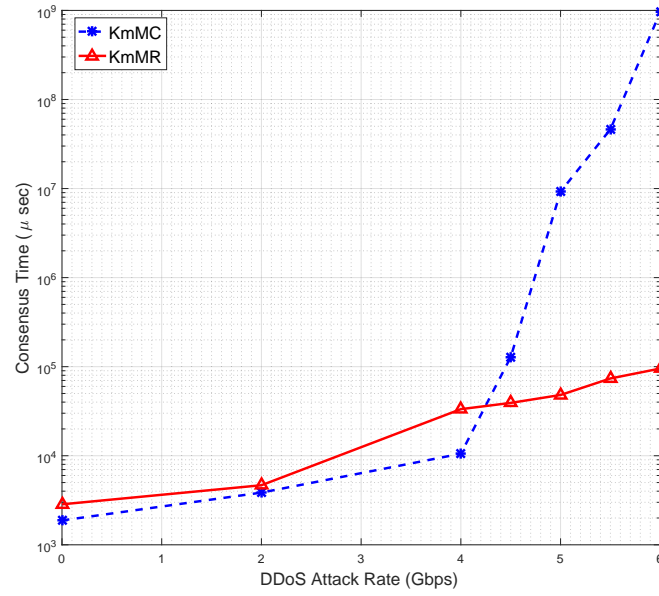
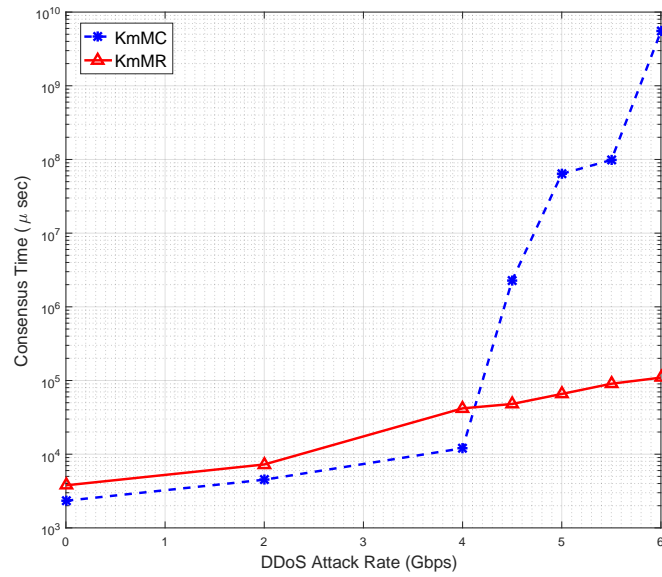Figure 9: Consensus time versus DDOS attack rate (*Scenario 2*, $n = 5$)

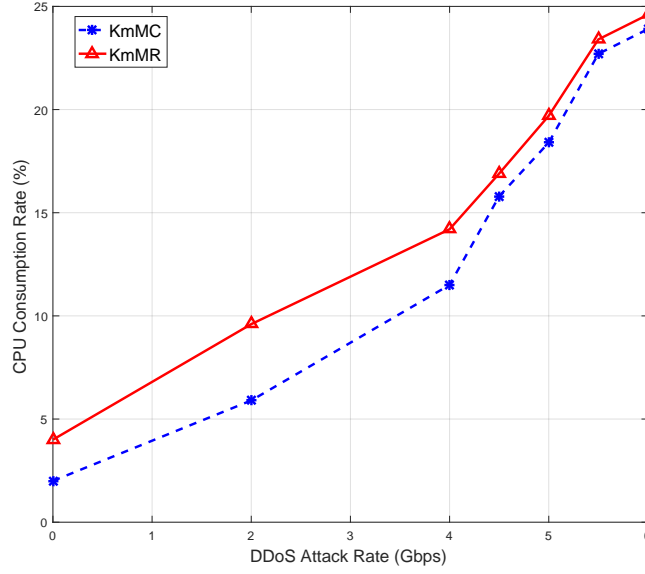Figure 10: Consensus time versus DDOS attack rate (*Scenario 2*, $n = 7$)

Figure 11: CPU consumption rate versus DDOS attack rate (*Scenario 2*, $n = 7$)

Fig. 10 illustrates the consensus time versus DDoS attack rate in the same environment as Fig. 9, but for a cluster of 7 master nodes. The same behavior is exhibited for $n = 5$ and $n = 7$ master nodes. However, for $n = 5$, consensus is established faster thanks to the smaller number of exchanged messages. As n increases, KmMC becomes more susceptible to DDoS attacks. Indeed, the rapid degradation of KmMC's performance starts at attack rate 4.1 Gbps for $n = 7$, compared to 4.25 Gbps for $n = 5$. Whereas, KmMR is able to establish consensus in a reasonable time, even for high attack rates.

Figs. 11-13 present the CPU, RAM and Bandwidth performances of the DDoS victim node, for *Scenario 2* and $n = 7$. When the DDoS attack rate is below 4.5 Gbps, KmMR uses as much or more resources than KmMC. This is expected since establishing a consensus in KmMR using BFT-SMaRt requires a larger number of messages exchange. However, for attack rates above 4.5 Gbps, KmMR and KmMC have almost the same level of resource utilization. Indeed, Raft starts to make changes in the cluster in order to regain its stability, resulting in higher resources consumption than usual.
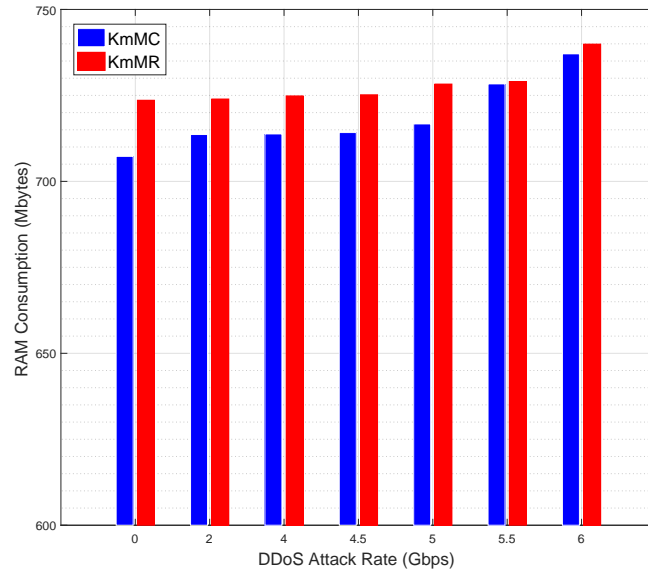
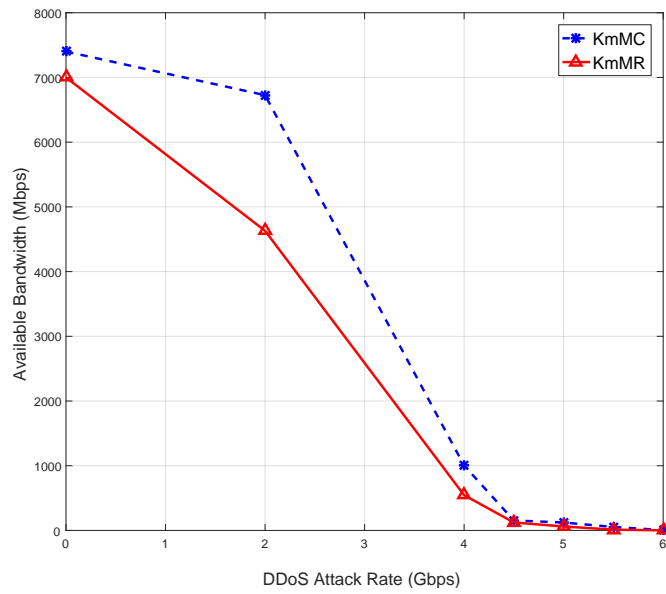Figure 12: RAM consumption versus DDOS attack rate (*Scenario 2*, $n = 7$)

Figure 13: Available Bandwidth versus DDOS attack rate (*Scenario 2*, $n = 7$)

6.3. Solution Limitations and Future Insights

When researchers proposed PBFT-like protocols, such as BFT-Smart, their main concern was to enhance the performance of BFT in fault-free cases, while maintaining properties of Liveness and Safety, when faults occur. BFT-SMaRt aims to be robust in terms of high performance in fault-free executions, and correctness when faults happen. According to our experiments, it is clear that BFT-Smart is capable of surviving in a small and partially uncontrolled environment, where Byzantine faults may occur. However, it would reach rapidly its limits in a larger network, mainly due to its heavy communication and limited scalability.

To reinforce resistance to malicious Byzantine faults, the notion of robust BFT protocols has been introduced by Aardvark, i.e., maintaining a constant and stable performance in the presence of few Byzantine faults [70]. Indeed, several improvements have been proposed, such as Aardvark [70], Spinning [71], and RBFT [72], in order to efficiently handle some worst-case malicious Byzantine behaviors. For instance, Aardvark tolerates nodes/replicas to expect a minimum acceptable throughput from the leader [70], while Spinning changes the leader with every batch of requests [71]. Finally, RBFT [72] proposed to maintain a constant performance during a fault event. It is demonstrated in [72] that Aardvark and Spinning performances are reduced by at least 78% in presence of a fault, whereas RBFT degrades by only 3%. This is due to RBFT's design, where $f + 1$ protocol instances are ran, but only one executes the received request.

Although interesting, the previous protocols would experience difficulties in managing inconsistency in large scale systems [73]. Indeed, this type of management is relegated to client nodes, although the reason to use a consistent BFT protocol is precisely to avoid this responsibility to clients. One of the promising solutions is to concurrently run independent processes aiming at achieving higher throughputs [74], which is the basic approach to implement scalable blockchain architectures.

Blockchain, by itself, is a BFT replicated state machine, where each state-update is a Turing machine with bounded resources. Unlike conventional BFT protocols where fault tolerance is realized among a small/medium group of nodes through rounds of message exchanges (votes and safety-proofs messages), blockchain achieves BFT

among a very large number of participants, where at each time period, only a single message (Proof-of-Work -PoW- message) is broadcast by a participant. Adopting known BFT mechanisms into blockchain has led to the proposal of hybrid solutions, such as Byzcoin [75], Bitcoin-NG [76], Casper [77] and Solida [78]. These approaches anchor off-chain BFT decisions inside a PoW chain or the other way around. For instance, Casper is a proof-of-stack (PoS)-based finality system, which overlays a PoW blockchain. By design, Casper allows to provide Safety and plausible Liveness, as well as protect the system against *long range revisions* and *catastrophic crashes* faults [77]. Moreover, innovative solutions in the age of blockchains, such as Honeybadger [79], Algorand [80], and LightChain [81], revisit the BFT setting with greater scalability and simplicity. Honeybadger is a demonstrative example of how BFT can build a blockchain cryptocurrency [79]. It can reach consensus within 5 minutes using 104 nodes. By design, Honeybadger requires prior setting of a fixed number of consensus nodes, which may be problematic in terms of targeted attacks that may either compromise the nodes or exclude them from the system. In contrast, Algorand, a PoS approach, can achieve better performance without having to select a fixed set of nodes beforehand [80]. Also, it is robust against malicious attacks, even from a malicious leader, and scales better for a large number of clients. Finally, in order overcome the low communication and storage efficiency, inconsistency and scalability problems encountered in existing blockchains, the authors in [81] proposed LightChain. The latter is a blockchain defined over a skip graph-based peer-to-peer distributed hash table overlay, which achieves consensus through Proof-of-Validation (PoV), i.e., a blockchain data is considered valid if its hash value is signed by a randomly selected number of validators [81]. It has been proven that LightChain is a fair, consistent, and communication/storage efficient blockchain.

In spite of its limits, it is clear that the implementation of BFT-Smart into Kubernetes is the first step into providing robustness to this popular Docker containers orchestration platform. As future work, one could investigate the integration of more sophisticated protocols to Kubernetes, such as the aforementioned ones, and test their robustness to malicious Byzantine behaviours. Testing can be realized through the BFT-bench framework introduced in [66].

7. Conclusion

With the increased importance of virtualization in cloud computing, Docker containerization is favored for its lightweight and efficient virtualization. This implies the emergence of new forms of architectures organizing cloud services in containers, ready to be instantiated in virtual and/or physical machines. Since the main objective is to guarantee service continuity, orchestrating these containers may seem challenging. Recently, Kubernetes has been adopted as the orchestration platform of Docker containers. Although efficient in managing containers, Kubernetes guarantees service continuity only in presence of non-Byzantine (crash) faults occuring within the system. In fact, the current replication protocol within Kubernetes "Raft" cannot handle Byzantine faults. In this paper, we propose a new orchestration platform capable of overcoming this limitation in Kubernetes. The KmMR platform, based on Byzantine replication protocol BFT-SMaRt, is presented. We detailed our approach to integrate the BFT-SMaRt library (written in Java) into Docker and Kubernetes (written in Golang). Then, we implemented a Kubernetes multi-master platform in an OpenStack-based cloud environment. The system is evaluated for two different scenarios, where initially the maximum number of tolerated faults is either reached or not, and for two orchestration platforms, KmMC and KmMR. The results show that the conventional approach KmMC is efficient and robust in a non-Byzantine and controlled environment, i.e. number of maximum tolerated faults is not exceeded. However, in a Byzantine and not fully controlled environment, KmMR guarantees the continuity of services, while KmMC collapses in front of severe Byzantine faults. In a such environment, KmMR resources consumption is typically stable, compared to KmMC. In future works, we will investigate the integration of more robust BFT protocols into Kubernetes, in order to ensure a better protection against malicious Byzantine faults.

References

[1] G. K. Thiruvathukal, K. Hinsen, K. Läufer, J. Kaylor, Virtualization for Computational Scientists, Computing in Science Engineering 12 (4) (2010) 52–61.

[2] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, R. Boutaba, Network Function Virtualization: State-of-the-Art and Research Challenges, IEEE Commun. Surveys Tuto. 18 (1) (2016) 236–262.

[3] D. Bernstein, Containers and Cloud: From LXC to Docker to Kubernetes, IEEE Cloud Computing 1 (3) (2014) 81–84.

[4] R. Peinl, F. Holzschuher, F. Pfitzer, Docker Cluster Management for the Cloud-Survey Results and Own Solution, J. Grid Comput. 14 (2) (2016) 265–282.

[5] R. Rizki, A. Rakhmatsyah, M. A. Nugroho, Performance Analysis of Container-based Hadoop Cluster: OpenVZ and LXC, in: Proc. 4th Int. Conf. on Information and Commun. Tech. (ICoICT), 2016, pp. 1–4.

[6] H. Zhang, G. Jiang, K. Yoshihira, H. Chen, Proactive Workload Management in Hybrid Cloud Computing, IEEE Trans. Network and Service Management 11 (1) (2014) 90–100. `doi:10.1109/TNSM.2013.122313.130448`.

[7] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, S. Tilkov, Microservices: The Journey So Far and Challenges Ahead, IEEE Software 35 (3) (2018) 24–35.

[8] S. Garg, S. Garg, Automated Cloud Infrastructure, Continuous Integration and Continuous Delivery using Docker with Robust Container Security, in: Proc. IEEE Conf. Multimedia Info. Process. and Retrieval (MIPR), 2019, pp. 467–470.

[9] R. Zhang, M. Li, D. Hildebrand, Finding the Big Data Sweet Spot: Towards Automatically Recommending Configurations for Hadoop Clusters on Docker Containers, in: Proc. IEEE Int. Conf. Cloud Eng., 2015, pp. 365–368.

[10] A. Sill, Emerging Standards and Organizational Patterns in Cloud Computing, IEEE Cloud Computing 2 (4) (2015) 72–76.

[11] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, J. Wilkes, Borg, Omega, and Kubernetes, Queue 14 (1) (2016) 10.

[12] Google, Kubernetes (2018).
URL https://kubernetes.io/

[13] C. Oliveira, L. C. Lung, H. Netto, L. Rech, Evaluating Raft in Docker on Kubernetes, in: Proc. Int. Conf. Syst. Science, Springer, 2016, pp. 123–130.

[14] D. Ongaro, J. K. Ousterhout, In Search of An Understandable Consensus Algorithm, in: Proc. USENIX Annual Technical Conf., 2014, pp. 305–319.

[15] A. Bessani, J. Sousa, E. E. P. Alchieri, State Machine Replication for the Masses with BFT-SMART, in: Proc. 44th Annual IEEE/IFIP Int. Conf. Dependable Syst. and Net., 2014, pp. 355–362.

[16] C. N. Copeland, H. Zhong, Tangaroa: A Byzantine Fault Tolerant Raft, scs.stanford.edu (2014).

[17] M. Correia, D. G. Ferro, F. P. Junqueira, M. Serafini, Practical Hardening of Crash-tolerant Systems, in: Proc. of the USENIX Conf. on Annual Technical Conf. (USENIX ATC), USENIX Association, Berkeley, CA, USA, 2012, pp. 41–41.

[18] R. P. Padhy, Docker Containers and Kubernetes: An Architectural Perspective (2018).
URL https://dzone.com/articles/docker-containers-and-kubernetes-an-architectural

[19] A. Moga, T. Sivanthi, C. Franke, OS-Level Virtualization for Industrial Automation Systems: Are We There Yet?, in: Proc. 31st Annual ACM Symp. Applied Comput., ACM, 2016, pp. 1838–1843.

[20] VMware vSphere Hypervisor (2019).
URL www.vmware.com/products/vsphere-hypervisor.html

[21] B. B. Rad, H. J. Bhatti, M. Ahmadi, An Introduction to Docker and Analysis of its Performance, Int. Journal of Comput. Sci. and Net. Security (IJCSNS) 17 (3) (2017) 228–235.

[22] A. M. Joy, Performance Comparison Between Linux Containers and Virtual Machines, in: Proc. Int. Conf. Advances in Comput. Eng. and Appl. (ICACEA), IEEE, 2015, pp. 342–346.

[23] V. T., Advantages of Docker (2015).
URL `jyx.jyu.fi/bitstream/handle/123456789/48029/1/URN%3ANBN%3Afi%3Ajyu-201512093942.pdf`

[24] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An Updated Performance Comparison of Virtual Machines and Linux Containers, in: Proc. Int. Symp. Perf. Analysis of Syst. and Soft. (ISPASS), IEEE, 2015, pp. 171–172.

[25] P. Sharma, L. Chaufournier, P. Shenoy, Y. Tay, Containers and Virtual Machines at Scale: A Comparative Study, in: Proc. 17th Int. Middleware Conf., ACM, 2016, p. 1.

[26] W. Li, A. Kanso, Comparing Containers versus Virtual Machines for Achieving High Availability, in: Proc. IEEE Int. Conf. Cloud Eng., 2015, pp. 353–358.

[27] A. Manu, J. K. Patel, S. Akhtar, V. Agrawal, K. B. S. Murthy, Docker Container Security via Heuristics-based Multilateral Security-conceptual and Pragmatic Study, in: Proc. Int. Conf. Circuit, Power and Comput. Tech. (ICCPCT), IEEE, 2016, pp. 1–14.

[28] CoreOS, Fleet Project (2014).
URL `https://github.com/coreos/fleet`

[29] Apache, Mesos Project (2014).
URL `https://github.com/apache/mesos`

[30] A. Luzzardi, V. Victor, Swarm: A Docker-native Clustering System (2014).
URL `https://github.com/docker/swarm/`

[31] K8s, Kubernetes Source Code (2014).
URL https://github.com/kubernetes/kubernetes/

[32] A. AWS, GoPro Reduces Compute Footprint by 70% Using Amazon ECS.
URL https://aws.amazon.com/solutions/case-studies/gopro-containers/

[33] M. Mazzara, N. Dragoni, A. Bucchiarone, A. Giaretta, S. T. Larsen, S. Dustdar, Microservices: Migration of a Mission Critical System, IEEE Trans. Services Comput. (2018) 1–1.

[34] A. Banerjee, K. K. Venkatasubramanian, T. Mukherjee, S. K. S. Gupta, Ensuring Safety, Security, and Sustainability of Mission-Critical Cyber–Physical Systems, Proceedings of the IEEE 100 (1) (2012) 283–299.

[35] N. Kratzke, P.-C. Quint, Understanding Cloud-native Applications After 10 Years of Cloud Computing: A Systematic Mapping Study, J. Systems and Software 126 (2017) 1–16.

[36] G. Sayfan, Mastering Kubernetes: Master the Art of Container Management by using the Power of Kubernetes, 2nd Edition, Packt Publishing, 2018.

[37] N. Bila, P. Dettori, A. Kanso, Y. Watanabe, A. Youssef, Leveraging the Serverless Architecture for Securing Linux Containers, in: Proc. IEEE 37th Int. Conf. Dist. Comput. Syst. Wrkshps. (ICDCSW), IEEE, 2017, pp. 401–404.

[38] CoreOS, Coreos ETCD (2018).
URL https://coreos.com/etcd/

[39] F. Cristian, Understanding Fault-tolerant Distributed Systems, Commun. of the ACM 34 (2) (1991) 56–78.

[40] L. Lamport, R. Shostak, M. Pease, The Byzantine Generals Problem, ACM Trans. Programm. Languages and Syst. (TOPLAS) 4 (3) (1982) 382–401.

[41] P.-L. Aublin, Towards Efficient and Robust Fault-Tolerant Protocols (in French), Ph.D. thesis, Université de Grenoble (2014).

[42] F. B. Schneider, Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial, ACM Comput. Surveys (CSUR) 22 (4) (1990) 299–319.

[43] M. Pease, R. Shostak, L. Lamport, Reaching Agreement in the Presence of Faults, Journal of the ACM (JACM) 27 (2) (1980) 228–234.

[44] L. Lamport, et al., Paxos Made Simple, ACM SiGACT News 32 (4) (2001) 51–58.

[45] R. Van Renesse, N. Schiper, F. B. Schneider, Vive la différence: Paxos vs. View-stamped Replication vs. Zab, IEEE Trans. Dependable and Secure Comput. 12 (4) (2015) 472–484.

[46] G. Bracha, S. Toueg, Asynchronous Consensus and Broadcast Protocols, J. ACM (JACM) 32 (4) (1985) 824–840.

[47] M. Castro, B. Liskov, Practical Byzantine Fault Tolerance and Proactive Recovery, ACM Trans. Computer Syst. (TOCS) 20 (4) (2002) 398–461.

[48] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, P. Verissimo, Efficient Byzantine Fault-Tolerance, IEEE Trans. Computers 62 (1) (2013) 16–30.

[49] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, T. Riche, Upright Cluster Services, in: Proc. 22nd Symp. Operating Systems Principles (SIGOPS), ACM, 2009, pp. 277–290.

[50] Y. Amir, B. Coan, J. Kirsch, J. Lane, Prime: Byzantine Replication Under Attack, IEEE Trans. Dependable and Secure Comput. 8 (4) (2011) 564–577.

[51] J. Sousa, A. Bessani, M. Vukolic, A Byzantine Fault-Tolerant Ordering Service For The Hyperledger Fabric Blockchain Platform, in: Proc. 48th Annual Int. Conf. Dependable Syst. and Net. (DSN), IEEE, 2018, pp. 51–58.

[52] X. L. Blake Mizerany, Y. Qin, The Raft Consensus Algorithm (2018).
URL https://raft.github.io//#implementations

[53] H. V. Netto, L. C. Lung, M. Correia, A. F. Luiz, L. M. S. de Souza, State Machine Replication in Containers Managed by Kubernetes, J. of Syst. Arch. 73 (2017) 53–59.

[54] J. Lim, T. Suh, J. Gil, H. Yu, Scalable and Leaderless Byzantine Consensus in Cloud Computing Environments, Inf. Syst. Frontiers 16 (1) (2014) 19–34.

[55] L. Perronne, Towards Efficient and Robust BFT Protocols (in French), Ph.D. thesis, Université Grenoble Alpes (2016).

[56] A. Souri, S. Pashazadeh, A. Navin, Consistency of Data Replication Protocols in Database Systems: A Review, International Journal on Information Theory (IJIT) 3 (2014) 19–32.

[57] GitHub, BFT-Smart Library (2018).
URL https://github.com/bft-smart/library

[58] L. Lamport, D. Malkhi, L. Zhou, Reconfiguring A State Machine, ACM SIGACT News 41 (1) (2010) 63–73.

[59] A. N. Bessani, M. Santos, J. Felix, N. F. Neves, M. Correia, On the Efficiency of Durable State Machine Replication, in: Proc. USENIX Annual Tech. Conf., 2013, pp. 169–180.

[60] J. Sousa, A. Bessani, From Byzantine Consensus To BFT State Machine Replication: A Latency-Optimal Transformation, in: Proc. 9th European Dependable Comput. Conf. (EDCC), IEEE, 2012, pp. 37–48.

[61] Golang, The Go Programming Language (2018).
URL https://golang.org/

[62] O. Sefraoui, M. Aissaoui, M. Eleuldj, OpenStack: Toward An Open-source Solution for Cloud Computing, Int. J. Computer Appl. 55 (3) (2012) 38–42.

[63] B. Schroeder, et al., Open Versus Closed: A Cautionary Tale, in: In NSDI, USENIX Association, 2006, pp. 239–252.

[64] Sanfilippo, Hping3 (2014).
URL `www.hping.org/hping3.html`

[65] B. Ops, Denial-of-Service Attack–DOS Using Hping3 with Spoofed IP in Kali Linux, BlackMORE Ops. BlackMORE Ops 17 (2016).

[66] D. Gupta, Towards Performance and Dependability Benchmarking of Distributed Fault Tolerance Protocols, Ph.D. thesis, Grenoble Alpes University (2016).
URL `https://tel.archives-ouvertes.fr/tel-01376741`

[67] D. Gupta, L. Perronne, S. Bouchenak, BFT-Bench: A Framework to Evaluate BFT Protocols, in: Proc. 7th ACM/SPEC Int. Conf. Perform. Eng. (ICPE), 2016, pp. 109–112. `doi:10.1145/2851553.2858667`.

[68] iPerf, IPerf (2018).
URL `https://iperf.fr/fr/iperf-doc.php`

[69] S. BISWAS, A Guide to The Linux Top Command (2018).
URL `www.booleanworld.com/guide-linux-top-command/`

[70] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, M. Marchetti, Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults, in: NSDI, Vol. 9, 2009, pp. 153–168.

[71] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary, in: Proc. 28th IEEE Int. Symp. Rel. Dist. Syst., 2009, pp. 135–144. `doi:10.1109/SRDS.2009.36`.

[72] P. Aublin, S. B. Mokhtar, V. Quéma, RBFT: Redundant Byzantine Fault Tolerance, in: Proc. IEEE 33rd Int. Conf. Dist. Comp. Syst., 2013, pp. 297–306. `doi:10.1109/ICDCS.2013.53`.

[73] E. Buchman, Tendermint: Byzantine Fault Tolerance in the Age of Blockchains, Master's thesis, University of Guelph (2016).
URL `https://atrium.lib.uoguelph.ca/xmlui/handle/10214/9769`

[74] R. Kotla, M. Dahlin, High throughput Byzantine fault tolerance, in: Proc. Int. Conf. Depend. Syst. and Net., 2004, pp. 575–584. `doi:10.1109/DSN.2004.1311928`.

[75] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, B. Ford, Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing, in: Proc. 25th USENIX Conf. Sec. Symp., USENIX Association, USA, 2016, p. 279–296.

[76] I. Eyal, A. E. Gencer, E. G. Sirer, R. V. Renesse, Bitcoin-NG: A Scalable Blockchain Protocol, in: 13th USENIX Symp. Network. Syst. Design and Implement. (NSDI 16), USENIX Association, Santa Clara, CA, 2016, pp. 45–59.

[77] V. Buterin, V. Griffith, Casper the Friendly Finality Gadget (2017). `arXiv:1710.09437`.

[78] I. Abraham, D. Malkhi, K. Nayak, L. Ren, A. Spiegelman, Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus (2016). `arXiv:1612.02916`.

[79] A. Miller, Y. Xia, K. Croman, E. Shi, D. Song, The Honey Badger of BFT Protocols, in: Proc. ACM Conf. Comp. and Commun. Sec. (SIGSAC), Association for Computing Machinery, New York, NY, USA, 2016, p. 31–42. `doi:10.1145/2976749.2978399`.

[80] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, N. Zeldovich, Algorand: Scaling Byzantine Agreements for Cryptocurrencies, in: Proc. 26th Symp. Op. Syst. Princ. (SOSP), Association for Computing Machinery, New York, NY, USA, 2017, p. 51–68. `doi:10.1145/3132747.3132757`.

[81] Y. Hassanzadeh-Nazarabadi, A. Kupçu, O. Ozkasap, LightChain: A DHT-based Blockchain for Resource Constrained Environments (2019). `arXiv:1904.00375`.
URL `https://arxiv.org/abs/1904.00375`