International Journal of Parallel, Emergent and Distributed Systems

ISSN: 1744-5760 (Print) 1744-5779 (Online) Journal homepage: https://www.tandfonline.com/loi/gpaa20

Incorporating the Raft consensus protocol in containers managed by Kubernetes: an evaluation

Hylson Netto, Caio Pereira Oliveira, Luciana de Oliveira Rech & Eduardo Alchieri

To cite this article: Hylson Netto, Caio Pereira Oliveira, Luciana de Oliveira Rech & Eduardo Alchieri (2019): Incorporating the Raft consensus protocol in containers managed by Kubernetes: an evaluation, International Journal of Parallel, Emergent and Distributed Systems, DOI: 10.1080/17445760.2019.1608989

To link to this article: https://doi.org/10.1080/17445760.2019.1608989

Published online: 26 Apr 2019.

Submit your article to this journal

View Crossmark data

Full Terms & Conditions of access and use can be found at
https://www.tandfonline.com/action/journalInformation?journalCode=gpaa20

INTERNATIONAL JOURNAL OF PARALLEL, EMERGENT AND DISTRIBUTED SYSTEMS
https://doi.org/10.1080/17445760.2019.1608989

Taylor & Francis
Taylor & Francis Group

Check for updates

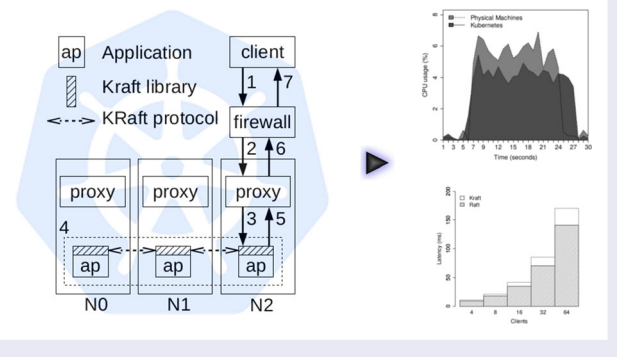
Incorporating the Raft consensus protocol in containers managed by Kubernetes: an evaluation

Hylson Netto [a], Caio Pereira Oliveira[b], Luciana de Oliveira Rech[b] and Eduardo Alchieri [c]

[a]Catarinense Federal Institute, Campus Blumenau, Brazil; [b]Department of Informatics and Statistics, Federal University of Santa Catarina, Brazil; [c]Department of Computer Science, University of Brasília, Brazil

ABSTRACT

Replication is a technique widely used to improve the reliability of applications. State machine replication is a special approach, where a set of computers are kept synchronised in the same state despite of failures that could occur in the system. The Raft algorithm can be used to implement a total order delivery protocol, delivering requests at the same order at all replicas, which is fundamental since in this approach all replicas must execute the same sequence of requests to present the same evolution in their states. Raft is easy to understand and implement, when compared to the Paxos algorithm. On the other hand, virtualisation can be seen as a technique that helps the development of reliable applications since it maintains each virtual machine (VM) isolated from the others. Virtualisation in data centres is changing from the traditional VMs to containers. In this context, this paper proposes KRaft, an incorporation of Raft in Kubernetes, a system that manages containers. After an evaluation of performance and resource consumption of KRaft, we found that it presents performance close to Raft executing on physical machines. Moreover, KRaft demands more network transmission while Raft executed in physical machines needs more processing power and memory.



ARTICLE HISTORY
Received 10 September 2018
Accepted 15 April 2019

KEYWORDS
Distributed agreement; state machine replication; virtualisation; containers

1. Introduction

The concept of virtualisation has appeared in the 60s and 70s when IBM developed operating systems that allow the use of virtualisation [1–3]. In the 80s, the hardware becomes cheaper than the previous

CONTACT Hylson Netto hylson.vescovi@ifc.edu.br

© 2019 Informa UK Limited, trading as Taylor & Francis Group

years and virtualisation was less important. However, in the 90s virtualisation received a lot of attention again, mainly by the appearance of the Java programming language. The benefits of the use of virtualisation include flexibility, security and costs reduction since it is easy to configure and to manage an application using virtualisation. On the other hand, it also brings some performance overhead to the applications.

Virtualisation can be seen as a technique that helps the development of reliable applications since it maintains each virtual machine (VM) isolated from the others. In fact, many works proposed its use to develop fault- and intrusion-tolerant applications [4–9].

Data centres make use of VMs to enable dynamic resource provisioning, providing a flexible and cost-effective (pay-per-use) resource sharing among users [10]. The access to these resources via Internet composes the *Cloud Computing*, defined by NIST as 'a model for enabling ubiquitous, convenient, on-demand network access to [...] computing resources [...] rapidly provisioned and released' [11]. Considering the importance to provide resource rapidly in data centres, *containers* have received a lot of attention, since they can provide faster resource allocation when compared with traditional VMs [12]. Among the available implementations of containers, Docker [13,14] is probably the most used. Facing the possibility of migration from VMs to containers [15], some companies founded the Cloud Native Computing Foundation (CNCF) [16] to drive the adoption of containers by the diverse cloud providers. The goals of CNCF include the creation of standards for container operation in clouds.

Google has a wide experience in using containers [17] and launched Kubernetes [13], that is an initial result from CNCF. Kubernetes is an open source cluster manager for Docker containers, it brings together knowledge from engineers of Borg [17], the container manager of Google. Kubernetes replicates containers to improve availability. When a container fails, Kubernetes recreates it from a predefined image. However, the state of a failed container is not restored. Applications can use external data volumes to maintain their state, but it is necessary to protect these volumes against failures. Furthermore, when providing state replication, applications have to handle concurrent access to these volumes.

Raft [18] is an algorithm derived from Paxos [19], that can be used to implement replicated state machines (RSM) [20] in local area networks (LAN). Raft is arguably easier to understand and implement than the Paxos algorithm, shedding more light on the study of consensus and replicated state machine protocols. Consensus is a fundamental building block to solve many practical problems that appear on reliable distributed systems. Through a consensus protocol, participants of a distributed system reach agreement in a single value despite of processes failures, allowing them to coordinate their actions in order to maintain state consistency and ensure system progress. In this way, by using Raft, it is possible to implement a total order delivery protocol [21] which allows that all requests sent to any replicated container are executed in the same order on all replicas (containers). This is a fundamental aspect for RSM implementation since it allows that all replicas present the same evolution in their states (i.e. ' synchronise' their states). Raft can be applied in Kubernetes at the application level, i.e. inside the containers.

Related work. Some previous work evaluated Raft [22,23] but measurements in terms of latency and throughput are still incipient. The author of Raft [22] has executed some preliminary tests of performance on which the system scales as more replicas are inserted in the system. However, latency is measured considering only one client, a scenario that is not the common one in data centres. Another work [23] repeated the Raft author's performance analysis, but its major measurements are also focused in the leader behaviour: neither the latency perceived by clients nor the throughput presented at the servers are mentioned.

This work extends a previous seminal performance evaluation of Raft in Kubernetes [24], which was focused on latency and throughput metrics and did not consider resources consumption. The latency and throughput of other consensus protocol in Kubernetes also was evaluated, like the *DORADO - orDering OveR shAreD memOry* protocol [25], that uses the etcd of Kubernetes as a shared memory. Although performance (latency and throughput) is a very important characteristic of some application,

the amount of resources required for its execution also is very relevant, mainly in a cloud computing environment since users of clouds pay according to the resources they allocate. Although clouds offer elastic resource allocation, measuring the resource consumption of RSM applications (according to different workloads) can provide useful parameters for users when contracting cloud providers. There are some initiatives that establish a relation between performance and resource consumption in clouds. For example, the performance of cloud computing services was measured according to network load and placement of VMs [26], the performance of scientific applications executed in the cloud was investigated [27] and also the adoption of containers for high-performance computing environments was considered [28]. However, further investigation is required to evaluate the performance and resource consumption in the context of RSM protocols executed in cloud environments. It is important to notice that our goal is not to evaluate different applications when executed in Kubernetes, but the costs associated with the RSM protocols. These costs could be seen as the overhead necessary to use an underlying RSM layer to support dependable execution of applications. Analysis regarding application costs already was exhaustively studied in the literature [27–31] and are orthogonal to this work since they refer to another layer.

Contributions. This paper extends previous studies, notably [24], by trying to answer the following question: *'How can we incorporate a consensus protocol in the Kubernetes architecture and what are the associated costs regarding performance and resources consumption?'* In answering this question, this paper presents the following contributions:

(1) It proposes a protocol for incorporating Raft in Kubernetes, resulting in the KRaft (Raft modified to run in Kubernetes). With this incorporation, we give more functionality to the containers managed by Kubernetes, i.e. requests sent to the system now could be executed at the same order at all containers, increasing the system with fault tolerance properties by implementing an RSM;
(2) It presents an evaluation of Raft when used to implement an RSM. We compare its execution in two scenarios: (a) on top of containers implemented by Docker and managed by Kubernetes; and (b) directly in physical machines. We found that the throughput presented by the system and the latency perceived by clients is similar in both environments. We also analysed the relation between some workloads and resources consumption, in order to characterise the behaviour of RSM implemented by Raft in the Kubernetes environment. Moreover, we also investigated the overhead required by Raft when compared with an application that does not replicate its state.

Paper organisation. The remaining of the paper is organised in the following way. Section 2 presents concepts about virtualisation and shows how Kubernetes manages containers. An overview about state machine replication and the Raft algorithm is presented in Section 3. Section 4 discusses the adaptions necessary to execute Raft in Kubernetes. Section 5 presents the evaluation of Raft in several scenarios. Section 6 discusses some lessons we acquired during the development of this work. Finally, Section 7 concludes the paper.

2. Virtualisation, containers and Kubernetes

The virtualisation concept appeared from the necessity of simulating the execution of computer instructions [1–3]. Virtualisation is useful, for example, to execute a software designed to a specific hardware in another different hardware, or even to execute a software before the target hardware is available. In 1974, it was estimated that if the real and simulated machine were the same (e.g. both are able to run software for the same hardware), it could be possible to run programs with a slow-down

about 20 to 1. Nowadays, we know that some simulated machines, known as VMs, can execute programs with the same speed as if the programs were running in the physical machines on which the VMs are hosted [32].

The Linux operating system has some components that enable virtualisation at the system level. Containers are VMs instantiated from static images. When a container turns off, its state[1] is lost. Usually, containers also do not maintain session data about specific clients. Consequently, containers are known as stateless VMs. Images of containers are usually small because only files that do not exist in its host are effectively stored in the container image. This is possible with the use of a layered file system, e.g. `aufs`. Therefore, the creation of containers is fast and resource provisioning becomes more efficient, in comparison to the traditional VMs [32]. The Linux kernel containment features is more known as LXC,[2] which gain popularity since Docker [13,14] extended it and became the most popular container implementation.

Google created Kubernetes [13] to provide a complete system for management of containers. Announced as an open source system, it is under active development by many engineers who build Borg [17], the container management system currently used by Google. Kubernetes inherits some concepts from Borg: for example, a web server produces logs and a log analyser reads them. These applications can be created in different containers, which is useful for updating each application separately. However, strongly coupled containers like these should stay on the same machine, in order to improve communication among containers. Borg has a feature called *Alloc*, which maintains containers on the same machine. Similarly, Kubernetes has a component called *POD* with the same goals of *Alloc*.

A Kubernetes cluster is composed of (virtual or physical) machines (Figure 1). Each machine is a *node*. A *POD* is a minimal management unit, and it can contain one or more containers.[3] PODs receive network addresses and are allocated to nodes. *Containers* that are inside a POD share resources, such as volumes where they can write and read data. Clients contact the cluster through a *firewall*, which distributes requests to nodes according to load balancing rules. The *proxy* receives requests from the firewall and forwards them to PODs. If a POD is replicated, the *proxy* component distributes the load, sending requests to one of the replicas. The *kubelet* manages PODs, containers, images and other elements in the node. It also sends data about the monitoring of containers to the main node, which will act when necessary.

The Kubernetes management components are in the *main* node. Components of Kubernetes interact via *REST* APIs. On the following, we highlight two of these components: *etcd* and kubectl. The tool *etcd* [33] implements the Storage component, which persists the state of the Kubernetes cluster. The *kubectl* is a command interface in which a human operator can interact with the cluster, e.g. to create PODs or to check for the health of the cluster. There are other management components in Kubernetes, but their description is out of the scope of this paper.

3. State machine replication and Raft

State machine replication [20] (SMR) is a technique applied to keep a set of computers synchronised in the same state in despite of failures that could occur in the system. This approach requires that the system has at least $n \geq 2f + 1$ replicas to tolerate up to f crash faults.

The idea behind this approach is that replicas start from a common initial state and, when the state of any computer is modified, the remaining computers of the group are also updated in the same way. To achieve this behaviour, every request is delivered to all replicas in the same order via some atomic broadcast protocol [21], what allows that replicas can execute the same set of operations at the same order. Moreover, to ensure that all replicas will reach the same final state, replicas must execute only deterministic operations.

The previously described behaviour is defined in terms of the following properties three properties [20]:

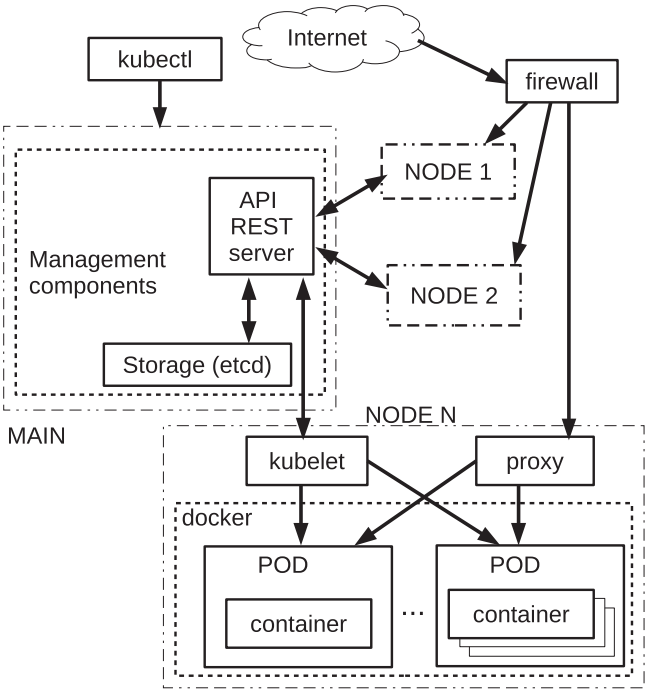


Figure 1. Kubernetes architecture.

- *Initial state:* All correct replicas start on the same state.
- *Determinism:* All correct replicas receiving the same input on the same state produce the same output and resulting state.
- *Coordination:* All correct replicas process the same sequence of commands.

The first two properties are not so hard to implement since they need that (1) all variables that represent the state of a replica start with the same values at all replicas and (2) all operations are deterministic. On the other hand, coordination need an atomic broadcast protocol and/or a consensus protocol, once they are equivalent [21].

Consequently, the implementation of an SMR needs a partially synchronous system model [34], on which the system is assumed to be asynchronous most of the time. However, there are windows on stability in a manner that messages are synchronously sent in the network. This is the weakest model of synchrony in which consensus is solvable deterministically.

Although SMR is an abstraction of high-level, many systems executed in clouds use replication algorithms to implement smaller actions like leader election and group membership. These services are frequently used to coordinate machines in data centers [35]. Despite the inherent replication cost, there are some examples in literature where SMR can be used to implement high-performance systems [36–38]. For example, some parts of the system can be provided with asynchronous replication [36] to benefit performance. Reordering requests can minimise the movements in the arm of hard disks and benefit performance from state machines backed by disks [37]. Another customisation is related to the workload of some systems: when read-only commands are left out of the ordering required by SMR, the system can provide the same latency of a non-replicated system [38].

Raft is a consensus algorithm that can be used to implement state machine replication. The Raft algorithm structure was created in a way that people can understand it easier than Paxos [19], the most popular algorithm of consensus. As a natural consequence, a high number of Raft implementations appeared [39]. The novel features carried in Raft refers to the way the leader communicates with

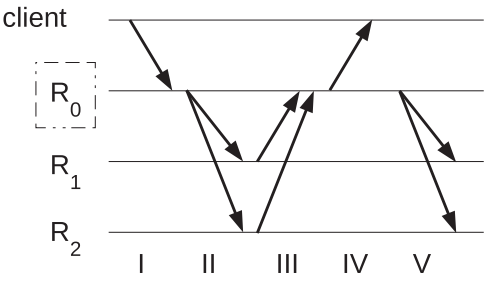


Figure 2. The Raft protocol (R_0 is the leader).

the replicas, the elections procedure and the membership changing. Firstly, updates on the replicated state are sent only from the leader to the replicas, which simplifies the management of the replicated log. As a consequence, only the leader accepts requests from clients. Requests sent to non-leader replicas are ignored. Second, randomised timeouts are used in leader elections, which solves conflicts in a simple and fast manner. It matches the load balance strategy implemented by Kubernetes, where containers are not differentiated among them and any container could be a leader. In fact, with the use of randomised timeouts to elect a leader, any container can be chosen leader in a non-deterministic manner. Finally, Raft uses the well-known quorums approach [40] to maintain the system operating during view changes. The majority of two different configurations overlap ensures the consistency of the replicated state while the configuration changes.

In Raft, one replica acts as a **leader** and the remaining replicas acts as **followers**. Raft operates in phases as follows (Figure 2):

- Phase I: first, a client send a request to a leader.
- Phase II: the leader sets an order to the request and presents the request to the followers.
- Phase III: followers accept the request and answer to the leader.
- Phase IV: the leader commits the request and answer to the client.
- Phase V: on the next event (a heartbeat or a new request order proposal), the leader sends a commit to the followers, which will also commit the request.

Replicas receive heartbeats from the leader. If after some time a replica did not receive any communication from the leader (by a heartbeat or by new ordered requests), it becomes a **candidate** replica and starts an election. The replica sends a `RequestVote` message to all replicas, which will answer with `Vote` messages. A replica did not accept a leader only if it has a more updated log entry than the proposed candidate and there is not yet a newly elected leader. A new leader is elected when it is accepted by a majority of replicas.

4. KRaft: Raft modified to run in Kubernetes

There are many Raft implementations available to download. We chose the *pontoon*[4] implementation that was developed using the Golang[5] programming language because of some specific reasons. Firstly, as a compiled language, it will provide a little executable code, which is favourable to keep the container size small. Our container with Raft has 53MB and is available in the Docker Hub[6] under the tag *KRaft*. Considering the available implementations of Raft in Go (at Raft site), we choose one we considered the most suitable to modify (ease of understanding is a relevant criteria: it is the design criteria of Raft). Another characteristic we considered is that Kubernetes is developed in Go, which could benefit the integration of the replicated state machine mechanism powered by Raft in Kubernetes. Integration is a technique that could make Kubernetes replicate states of any container in a transparent way [41]. Integrating a component in a platform is more easily when both are made under the same

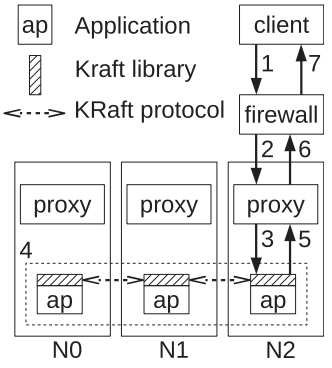


Figure 3. KRaft architecture.

technology [25]. In fact, *etcd* (Section 2) uses Raft (written in Golang) to make the Storage component of Kubernetes reliable, distributing its storage in other machines.

Considering the reasons above, we forked the *pontoon* implementation and created **KRaft**, a Raft ready to be executed in Docker containers at Kubernetes. Two modifications were done in Raft to enable its execution in containers managed by Kubernetes: the replica discovery mechanism and the acceptance of requests by any replica.

(1) *Replica Discovery Mechanism:* Each replica in Raft have to know the set of replicas that are joined in the current system configuration view. We changed the initialisation procedure of Raft to get the replicas list from the API Server of Kubernetes. This action is necessary because containers receive a dynamic IP address when they are instantiated. This usually does not happens in other SMR systems [42] because it is a common assumption to consider that servers have static network addresses. Moreover, replicas periodically query the API Server to keep updated the configuration view, adding or removing replicas that were changed by Kubernetes. These changes can happen by human operation (for example, with the goal of change the number of replicas) or automatically because of the Kubernetes monitoring (if it detects the failure of some container, which will result in its destruction and the creation of a new one).
(2) *Acceptance of Requests by any Replica:* The second modification refers to the acceptation of requests. In the original Raft, only the leader accepts requests (Section 3). Non-leader replicas are expected to discard requests received from clients. In Kubernetes, there is a load balance at the node level that distributed requests among replicated containers through the `proxy` component (Section 2). To match the load balance requirement, we modify the replica behaviour as follows: when a non-leader replica receives a request, it sends this request to the leader, waits for its execution, and forwards the answer to the client (see below).

KRaft Request Processing Phases/Steps.

KRaft runs in Kubernetes as a library together with the application (Figure 3). The client sends a request to a proxy through the firewall (steps 1 and 2). When a request is received at the proxy (step 2), it acts as a load balancer and forwards (step 3) the request to some container (replica). Replicas communicate among them using KRaft protocol (step 4) to coordinate and establish the order for the request execution. After that, the request is executed against the application state and the container which received the request answers the client again through the proxy and the firewall (steps 5, 6 and 7).

Depending on the proxy's choice on step 3, the system will present two different behaviours. In case the request is forwarded to the leader container, KRaft protocol proceeds as follows (Figure 4):

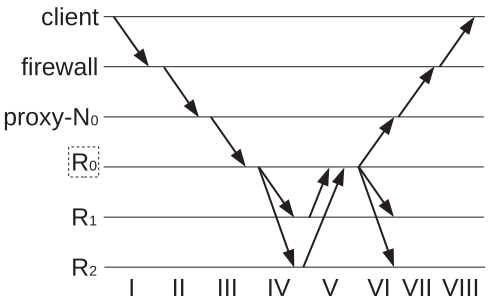


Figure 4. KRaft execution: request forwarded to the R_0 leader node.

- Phase I: first, the client send the request to the Kubernetes cluster.
- Phase II: the request is forwarded by a firewall to a node (N_0 in the example).
- Phase III: inside the node, the proxy forwards the request to a replicated container R_0 that is acting as a leader in the KRaft protocol.
- Phase IV: the leader R_0 assigns an order to the request and send it to the other replicas.
- Phase V: replicas accept the ordered request.
- Phase VI: the leader R_0 executes the request, reply to the client and sends commit to the other replicas.
- Phases VII e VIII: the reply is forwarded to the client through the proxy and the firewall.

Otherwise, in case the proxy forwards the request to a non-leader replica (container), this replica will acts as a client and forwards the request again to the leader replica. This could happen because the leader is located in only one of the nodes on which KRaft has the replicated containers and the proxy could forward the request to a node that does not contain the leader container. Moreover, a node where the leader is placed could also hosts other replicas of KRaft than the leader.

In case a non-leader replica receives the request, the KRaft protocol needs two additional steps (III-b and VI-b) and executes as follows (Figure 5):

- Phase I: the client send the request to the Kubernetes cluster.
- Phase II: the request is forwarded by a firewall to a node (N_1 in the example).
- Phase III: inside the node, the proxy forwards the request to a replicated container R_1 that is not acting as a leader in the KRaft protocol.
- Phase III-b: the non-leader container R_1 acts as a client of the system and sends the request to the leader container R_0 (dotted line).
- Phase IV: the leader R_0 assigns an order to the request and send it to the other replicas.
- Phase V: replicas accept the ordered request.
- Phase VI: the leader R_0 executes the request and sends commit to the other replicas. In addition to the commit message, the non-leader replica R_1 receives the answer since it is acting as a client (dotted line).
- Phase VI-b: the non-leader replica R_1 replies to the client.
- Phases VII e VIII: the reply is forwarded to the client through the proxy and the firewall.

Applications that could benefit from the proposed solutions

This paper proposed KRaft, a Raft consensus protocol modified to run in the containers orchestrator Kubernetes. This section discusses some practical aspects and characteristics related to the use of this

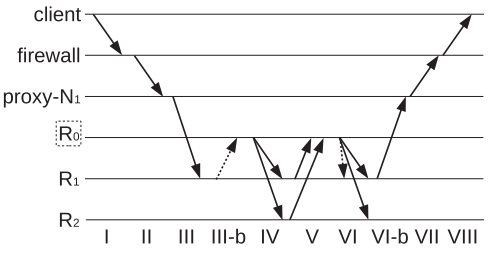


Figure 5. KRaft execution: request forwarded to the R_1 non-leader node (R_0 is the leader).

integration in the deployment of replicated applications. In summary, first it discussed the use of SMR in the development of applications and, finally, containers security and isolation issues are presented.

SMR and applications. Many applications can benefit from the proposed system, mainly critical applications that must ensure availability for the users. Availability entails tolerating component failures and is typically accomplished with replication. Many systems still use the primary backup strategy or similar protocols [43], which have to use a conservative timeout to reduce the possibility of asynchronous events that could lead to false failures suspicions. Another drawback of this strategy is the state transfer overhead, which could be prohibitive for applications with large states.

Following a different approach, SMR ensures data availability and consistency across replicas by using a consensus protocol (like KRaft) to implement an ordering protocol that ensures all replicas deliver and execute requests in the same order. Consequently, all replicas evolve synchronised, passing through the same states. This approach is a generic and well-known technique to implement fault-tolerant systems. In fact, a state machine replication system can be developed to be completely transparent for the applications [44,45], leaving it free from any modifications. Moreover, in this case, a programmer can focus on the inherent complexity of the application while remaining oblivious to the difficulty of handling replica failures or state consistency.

To ensure linearisability [46] (the consistency criteria for SMR), operations executed at the replicas must be deterministic, ensuring that they will produce the same reply and evolve through the same states. This limitation can prevent the use of this approach for some applications. Fortunately, techniques are being studied to handle non-determinism of replicated applications [47,48].

Regarding performance, although the SMR protocols impose some overhead for the applications (and potentially can preclude its use for some applications that need a very low latency – e.g. real-time applications), mainly its consensus protocol, many recent studies proposed solutions that exploit the applications semantic to improve the overall system performance (e.g. [38,49–51]).

Finally, the SMR approach has been successfully used in many large online services. Notable examples are Google's Chubby [52], Google Spanner [53], Windows Azure Storage [54], Scatter [55] and Apache Zookeeper [56]. Moreover, the emergence of blockchain and cryptocurrencies technologies brought a lot of attention to SMR protocols (e.g. [57–60]).

Containers security and isolation issues. Security issues are frequently pointed out as a weakness for hosting applications in containers. However, there are proposals for improving the security of hosted applications concerning the isolation between the containers. SCONE is a mechanism that uses a trusted computing base (TCB) to increases confidentiality and integrity of containerised applications [61]. Moreover, SCONE does not require any changes to the hosted application. Another effort is the Atomic project [62], a practical solution to improve isolation between containers. This solution uses the SELinux to increase the protection of both host and containers.

5. Evaluation

In this section, we assess the execution of Raft (the original Raft protocol executed in physical machines) and KRaft (our modified Raft protocol to run in Kubernetes previously presented at Section 4) under different conditions. In summary, our main goal is to verify if KRaft presents a performance similar to Raft and identify the resources usage for each protocol. This conjecture is supported by the evidence that applications hosted in containers can execute with similar performance of their executions directly in physical machines [32,63].

Moreover, as applications consume resources for their executions, these aspects also are important and must be investigated. Special interesting, in the cloud environment this is very relevant because users pay to the providers based on how much resources their applications require.[7] In this evaluation, we measure the amount of CPU, memory and network that Raft and KRaft require during their executions. These metrics are usually considered to measure resource consumption [12,28,63,64]. The first two characteristics (CPU and memory) determine which kind of machine is required to execute the application, while the last one (network) can reveal the minimal amount of bandwidth required to maintain replicas synchronised, specially when more clients access the system simultaneously.

Additionally, we also evaluated the overhead to maintain a service replicated across containers by comparing the execution of KRaft and a non-replicated application. State replication usually significantly impacts the system performance, but this is acceptable since this approach allows the creation of high available services [36–38].

5.1. Experiments setup

The experimental environment was configured with four machines Intel i7 3.5GHz, QuadCore, cache L3 8MB, 12GB RAM, 1TB HD 7200 RPM, connected via LAN Ethernet 10/100 MBits. The operating system used at each machine was Ubuntu Server, version 14.04.3 with kernel 3.19.0-42. We used Kubernetes version 1.1.7 and Docker 1.9.1. Containers that were allocated in distinct physical machines communicate through a virtual network implemented by Flannel.[8]

KRaft was executed in Docker containers managed by Kubernetes and Raft was executed directly on the same physical machines where Kubernetes is installed. Three replicas of Raft were placed on distinct machines to tolerate one replica failure. Kubernetes also always allocated containers in different machines, according to its default policy. This placement approach improves system availability since it enables the system to tolerate crash faults of the underlying hardware.

Resources were measured using the Versatile Resource Statistics Tool (`dstat`),[9] which is available in many Linux distributions. The `dstat` was started immediately before the creation of the KRaft containers and stopped just after the experiment execution. On the same way, in the physical machines context the `dstat` was started before Raft replicas started and was stopped some seconds (approximately 5 s) after all clients received answers. The time of observation for each execution stays around 30 s.

For the KRaft protocol, all consumption measurements were done on the first machine that served as a node of the Kubernetes cluster. The same machine was used for measurements in the case of Raft. Notice, however, that the results reported here represent the consumption of only one machine and not the total amount required by all machines. Consequently, considering the application replicates its state at n replicas, the total amount of resources required in the executions is approximately the values reported here multiplied by n.

The source-code of KRaft, Raft and the non-replicated application are available in GitHub[10] platform. These applications used in the experiments do not execute any operation, i.e. they implement an 'empty' service and only the request/response sizes can be configured. These benchmarks are commonly used to evaluate this kind of application [42] and we used the configuration 0/0 to assess only the impact caused by each approach. At the end of each execution, we checked the final state of

replicas by generating a *hash* of the *log* created by Raft or KRaft, according to the considered environment. This *hash* always was equal in all replicas, indicating that they executed the same sequence of operations.

To simulate multiple clients accessing the system, we used the Apache HTTP server benchmarking,[11] which is also available in common Linux distributions. The clients are located in the main machine of Kubernetes. We variate the number of clients from 4 up to 64, doubling the number of clients on each step. For each experiment, in spite of the number of clients, the system was exposed to 8000 requests. This mean that in a *n*-clients configuration, each client sent 8000/*n* requests to the system. For example, when the system operated with 4 clients, it means that each client sent 2000 requests.

5.2. Results and discussion

This section reports the results and discusses the main aspects that were gathered from the experiments. The results are divided into four classes: (1) application performance (throughput and latency); (2) application resources consumption (CPU, memory and network); (3) a performance comparison of KRaft to a non-replicated application, from where it is possible to assess the overhead imposed by the state replication and (4) an assessment of the impact of failures in the system.

5.2.1. Latency and throughput

Table 1 and Figure 6 present performance results for clients accessing both Raft and KRaft. The table shows that the latency mean measurements presented low standard deviation for all configurations. As expected, Raft performs better when running in physical machines and the overhead of KRaft over Raft in terms of latency increases as more clients access the system. However, KRaft has an overall efficiency around 82% of Raft, in terms of latency (last column of the Table 1). As more clients access the system, latency increases for all measured environments.

Throughput of KRaft grows up to around 376 requests per second, while Raft achieves 454 requests per second. Throughput gets higher as more clients enter in the system (Figure 6(b)), reaching the peak throughput when there are more than 8 clients. The throughput was calculated by dividing the total amount of requests executed (8000) by the total time demanded for its execution.

Table 1 also presents the results for an application (in Kubernetes) that did not replicate its state. As expected, since no coordination is needed, the performance is better in these cases but the system properties could be impaired by a single failure. The costs for accessing a non-replicated system and the overhead for replication are further discussed at Section 5.2.3.

Table 1. KRaft, Raft and non-replicated application measurements.

	KRaft			Raft			Non-replicated application			
	Lat.			Lat.			Lat.			
Clients	AV (ms)	SD (ms)	Thr. (req/s)	AV (ms)	SD (ms)	Thr. (req/s)	AV (ms)	SD (ms)	Thr. (req/s)	Latency Raft/KRaft (%)
4	11	1.2	114	9	1.0	141	2	0.4	2439	81.8
8	21	1.9	208	18	1.5	253	3	0.6	2973	85.7
16	42	2.6	377	35	2.2	453	5	0.6	2991	83.3
32	85	4.4	376	70	3.7	454	10	1	3051	82.4
64	170	9.3	375	141	7.7	452	21	1.6	2982	82.9

Notes: The table shows the latency (Lat.) perceived at clients and the throughput (Thr.) presented by servers. For latency, it is presented the average (AV) and the standard deviation (SD).

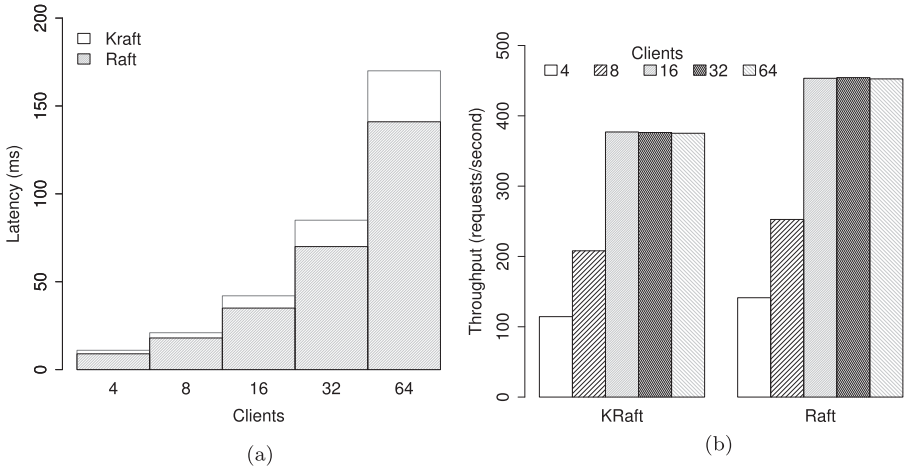


Figure 6. KRaft and Raft performance. (a) Latency and (b) throughput.

5.2.2. System resources consumption

This section analyses the results for memory, CPU (processors) and network consumption for both KRaft and Raft during 30 s of their execution (approximately, the time demanded to execute the 8000 requests).

5.2.2.1. Memory. Figure 7 presents the amount of memory used to execute each protocol. All graphics about memory are plotted discarding initial memory usage at physical machine or inside the container, because we want to show the behaviour of the consensus algorithm during the experiments execution. Additionally, this approach results in graphics of memory, CPU usage and network with the same measurement referential (the zero point at *Y* axis).

The memory consumption of Raft and KRaft are very similar since they execute the same algorithm for request ordering and execution. A statistical test (*t* of Student) indicates that the memory curves are not significantly different, although in Figures 7(c,d) Raft memory consumption depicts slightly higher values than KRaft. It has been shown that operating system level virtualisation has a memory consumption similar to the memory consumption in the physical machine [28,63]. After the first 5 s of monitoring, as more clients access the system, the memory consumption increases until the end of the experiment mainly because the log grows while requests are received and stored.

5.2.2.2. CPU (Processors). Monitoring the CPU usage shows that Raft uses CPU with more intensity than KRaft (Figure 8). However, Raft can execute the client's requests in less time than KRaft, corroborating results of latency presented in Figure 6(a). Container-based systems obtain execution times very close to native systems [28,63], which lead us to suppose that our modification in KRaft (Section 4) could have extended the processing time of KRaft.

Increasing the number of clients accessing the system, CPU consumption does not change significantly. While the clients are beings activated, CPU consumption remains near to zero. As clients start to send the requests, the nodes work to execute them increasing the CPU consumption. After a request has its order defined by the leader, the client can receive the answer because the request will be eventually executed by the replicas. As clients send its last requests, at the end of the experiment CPU consumption returns to zero.

5.2.2.3. Network. Network behaviour is illustrated in Figures 9 and 10. Sent and received data were very similar in KRaft, while in Raft the receiving of data was lower than the sending of data. Although network communication is similar in systems virtualised at the system level and physical

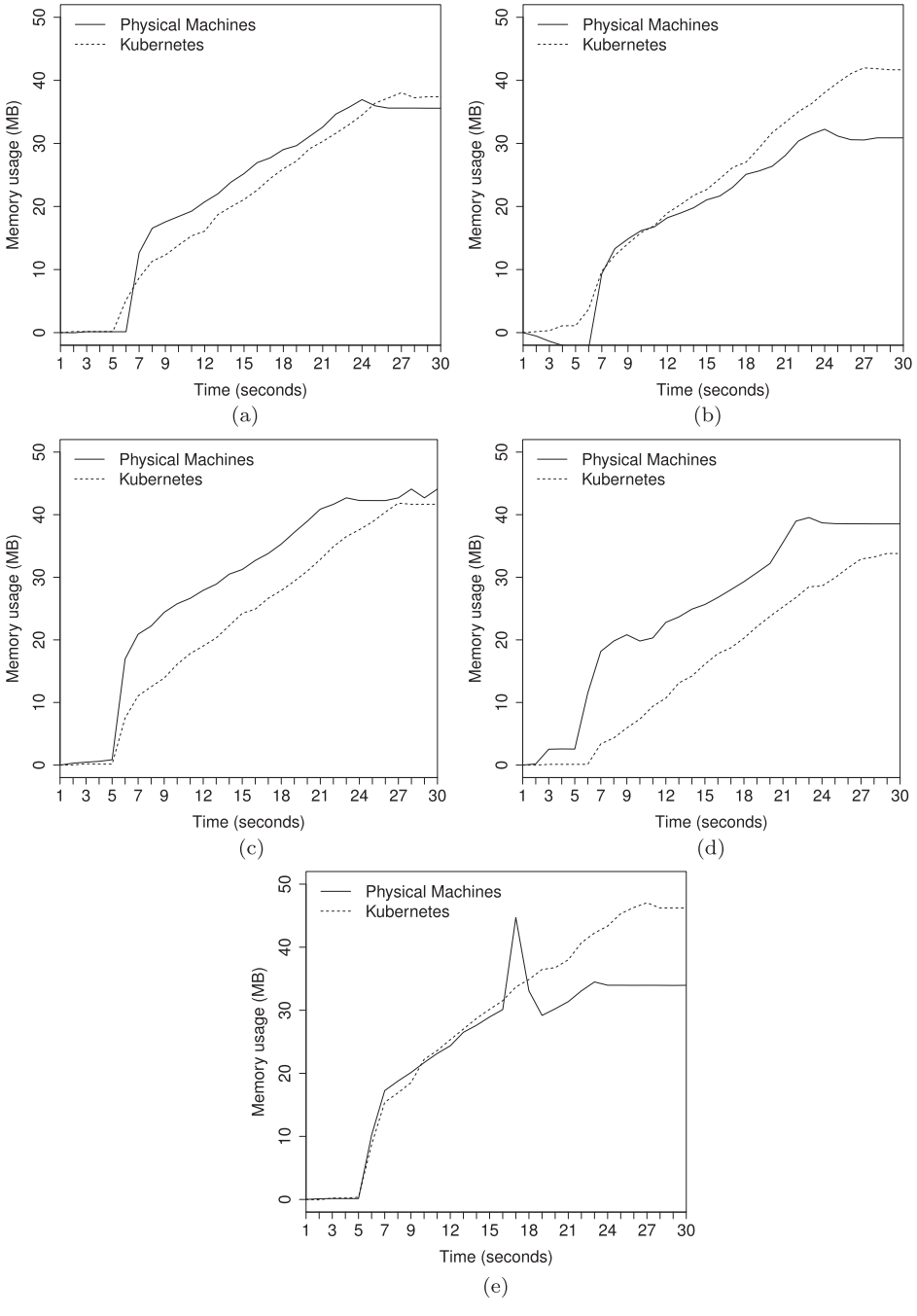


Figure 7. Memory usage of KRaft and Raft for a different number of clients. (a) 4 clients, (b) 8 clients, (c) 16 clients, (d) 32 clients, (e) 64 clients.

machines [28,63,64], a hypothesis about this network consumption difference is that our modifications in KRaft (Section 4, about forwarding requests to the leader and periodically querying the API Server of Kubernetes) generated this overhead detailed in Figure 10 for the case of 64 clients. In general, the total amount of network required was not high since KRaft used a mean bandwidth of 3.2 MB/s to operate. Raft required approximately 2.5 MB/s, which is 78% of the network used by KRaft.

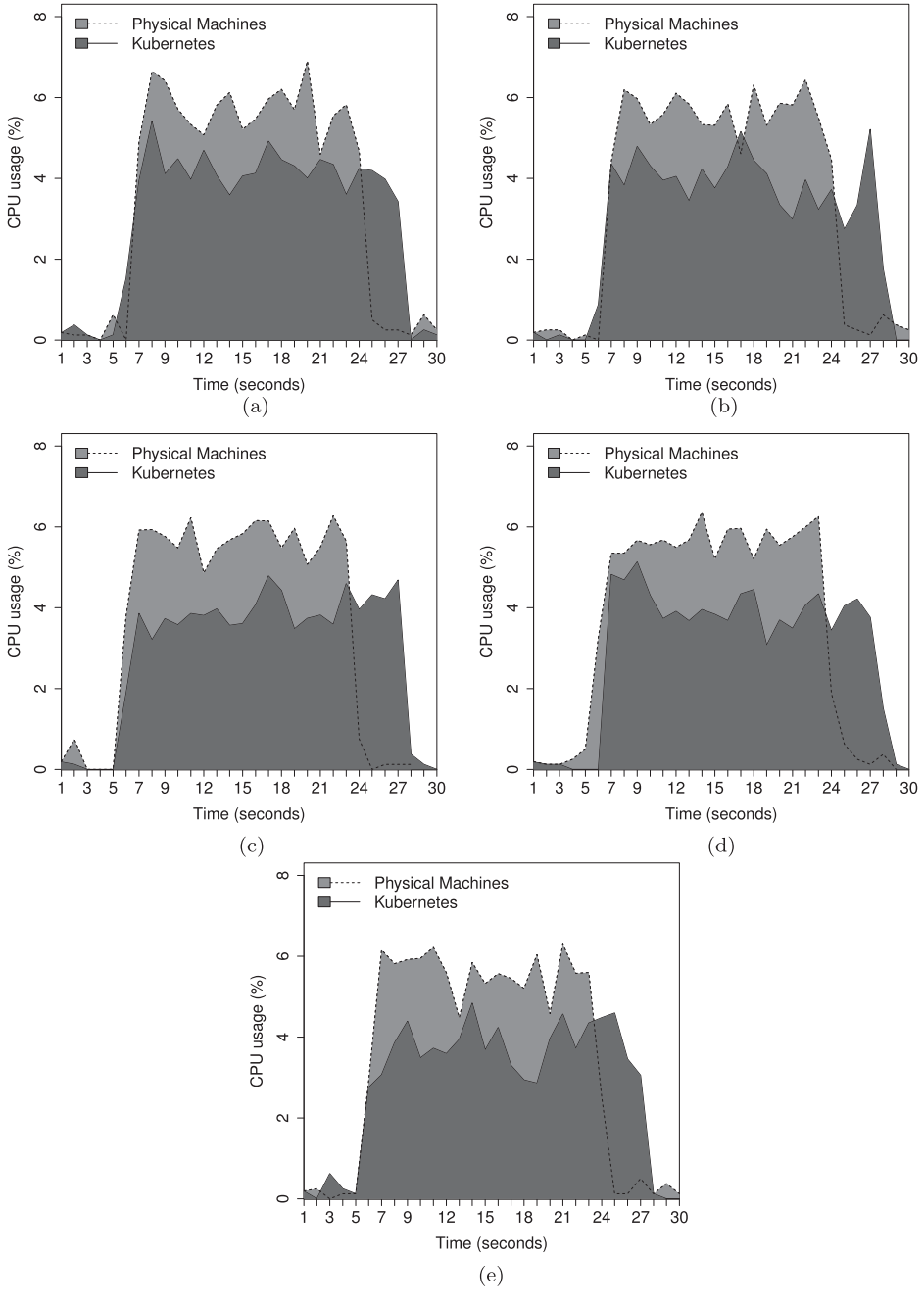


Figure 8. CPU usage of KRaft and Raft for different number of clients. (a) 4 clients, (b) 8 clients, (c) 16 clients, (d) 32 clients, (e) 64 clients.

Moreover, Figure 9 shows that network sending and receiving of data were not impacted by the amount of users contacting KRaft or Raft. In fact, data sending seems to be equal (in the sense that they have the same drawing) in Raft and KRaft. The behaviour of the data receiving also have the same shape for KRaft and Raft, only differing by the scale of the measured values. Similar to the CPU behaviour, network consumption increases when clients are being activated and start to send the requests. After requests are answered to the clients, network traffic decreases to zero.

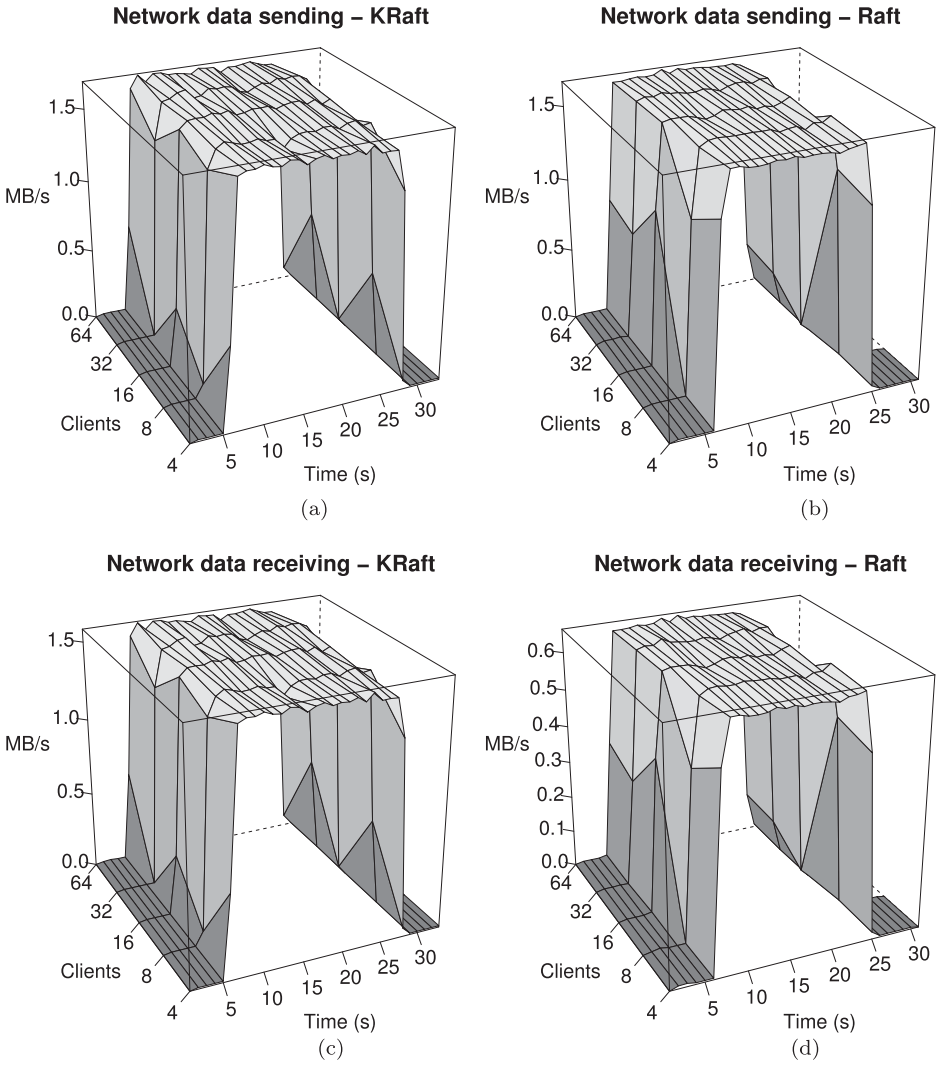


Figure 9. Network sending (top) and receiving (bottom) for KRaft and Raft. (a) KRaft sending, (b) Raft sending, (c) KRaft receiving, (d) Raft receiving.

5.2.3. Replication overhead

Figure 11 presents the KRaft overhead in the latency perceived by the clients when compared to an application that did not replicate its state (both configurations were executed in Kubernetes). As expected, the figure shows that KRaft cost is higher because it needs to execute the consensus protocol among the replicas to keep state consistency, what is not necessary for a non-replicated system. Moreover, a non-replicated system can also offer a higher throughput than a replicated system, as presented in Table 1.

However, the benefits of high availability and fault tolerance provided by KRaft can compensate this overhead for many critical system where availability is crucial. Moreover, replicated state machines can act as the basis of high-performance systems [36–38] if a careful engineering is applied. Some practices to provide the improvements are optimisations such as compaction of transmitted data [36], modifications in the internal structure of the log [37] or speculative execution and partitioning of the consensus instances [38].

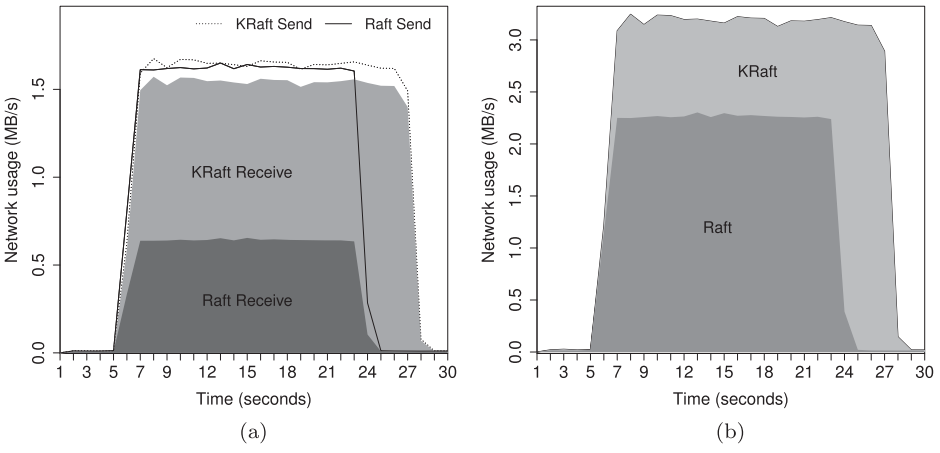


Figure 10. Network usage of KRaft and Raft: case of 64 clients. (a) Detailed send/receive consumption, (b) Total consumption.

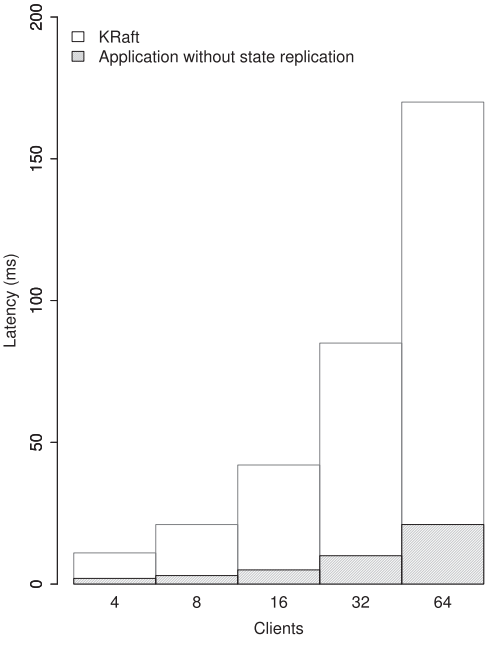


Figure 11. KRaft compared to a non-replicated application.

5.2.4. Impact of failures in the system

This section assesses the impact of failures in the performance of KRaft. The presented evaluation considers the normal case execution without failures as baseline to analyse the two possible scenarios with failures in the system: non-leader failure – a failure in a non-leader replica is masked by the other replicas in the system, i.e. the voting mechanisms used in the protocols will not consider the message from the failed replica; and leader failure – this is the worst scenario that significantly impacts system performance since a leader failure prevents the progress of the system, i.e. no order is assigned for the requests until the failure is detected (for this it is necessary to wait for a timeout) and a new leader is chosen to resume the protocols.

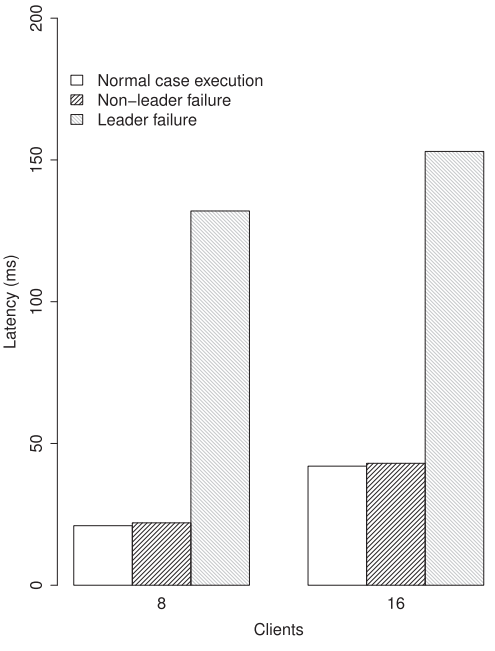


Figure 12. Impact of failures in KRaft.

Figure 12 presents the results for the execution of KRaft in the previously described scenarios, considering 8 and 16 clients accessing the system. Performance of KRaft is not affected by failures in non-leader replicas because KRaft algorithm use quorums (voting mechanisms) to get answers from faster (in this case the correct) replicas. In this case, the KRaft ordering protocol makes progress by using the messages exchanged by the correct replicas and no leader change is necessary.

However, performance is significantly impacted by a leader failure. Figure 12 shows that latency increases abruptly in this case. This behaviour happens because it is necessary to wait for a timeout (configured as 100 ms in these experiments) to detect the leader failure, i.e. if the request is not ordered within a defined time, then replicas suspect that the leader is faulty. Afterwards, it is necessary to choose a new leader that will resume the protocols by proposing an order for the requests not ordered yet. The right choice of a timeout is fundamental for performance in case of leader failure: a tight timeout may lead to false suspicion of leader failures, in this case replicas will chose another leader delaying the ordering protocol; a large timeout will delay the suspicion of a failed leader impacting performance in case of leader failures.

6. Lessons learned

During the development of this work, some important knowledge was acquired about the incorporation of consensus and SMR protocols in the Kubernetes architecture. This section discusses some of the main aspects that we realised during this work.

6.1. Load balancer vs. state machine replication

Load balancing is a feature designed to provide a better resource usage among the available replicas of the application. However, when a leader-based consensus protocol like Raft is used to implement a state machine replication strategy, the most efficient policy should be the forwarding of all

request to the leader. The internal load balancing provided by the proxy component of Kubernetes distributes requests among replicated containers, which lead us to create the ability of KRaft to deal with requests received by follower replicas (Figure 5). A more effective solution could be make the firewall to deliver the request straightly to the KRaft leader container, bypassing the internal proxy component. Kubernetes already has some features like Ingress[12] which enables this kind of implementation.

6.2. Replication costs

Many containers may operate inside the same operating system, meanwhile each traditional virtual machine provides a different operating system. Consequently, the use of traditional VMs demands much more resources (e.g.: memory and CPU) than containers. If the target application can be replicated without the requirement of operating system diversity (which is a common assumption when considering crash fault tolerance), it is possible to make a better usage of the machine resources. Containers share resources efficiently since they are a system-level virtualisation, and there are broadly available implementations like Docker. Considering the cloud computing paradigm of pay-per-use, it means that the nodes of the cluster (physical or virtual) could be used in a more efficient way, since many small containers (with different or replicated applications) could operate inside the same node, demanding less resources.

6.3. Consensus algorithms and virtualisation

The performance presented by Raft (physical machines) and KRaft (containers) are very similar. As virtualisation allows multiple applications running inside the same physical machine, resources are better used because the resources are shared among these applications instead of be totally dedicated to just one of them.

Moreover, when the fault tolerance threshold (i.e. the number of failures tolerated in the system) gets higher than the number of available physical machines, more than one replicated container could be allocated on top of the same physical machine. This allocation maintains the properties of fault tolerance since the containers are isolated among themselves and, at the same time, reduce resources consumption once some of them are shared among the containers (e.g. the operating system).

7. Conclusions

This paper presented an evaluation of the Raft consensus algorithm running in Docker containers and orchestrated by Kubernetes, a container management system. We found that the performance (both, throughput and latency) of KRaft (our modified implementation of Raft which enables its execution in Kubernetes) is similar to the execution of Raft in physical machines. However, the use of virtualisation facilitates the deployment and management of the replicated system, what justifies the use of containers and architectures like Kubernetes.

The resources consumption (CPU, memory and network) also does not presented relevant differences. Interesting, Raft in physical machines used more CPU than in Kubernetes, while the network consumption was the opposite (mainly, the amount of data received at the hosts). This behaviour is explained by the need of exchange data for containers management in Kubernetes. Memory consumption was similar because we executed the same consensus algorithm in the two environments (physical and virtual).

We also found that KRaft has significant overhead to replicate the state of the application hosted in containers when compared to a non-replicated service. This behaviour was expected since it is well-known that protocols and algorithms for state replication have a higher cost than a non-replicated service. This overhead is compensated by the service that is provided to the applications, i.e. it can be feasible for applications that require high availability.

Notes

1. State can be defined as all data stored inside the container from the time it was instantiated from the image.
2. https://linuxcontainers.org
3. Most of operations in Kubernetes manipulate PODs. However, we opted to refer the term *container* instead of POD on the following sections of this paper because the first one is a well-known term.
4. github.com/mreiferson/pontoon
5. golang.org
6. https://hub.docker.com/r/caiopo/raft/tags/
7. See, for example, aws.amazon.com/ec2/pricing/
8. coreos.com/flannel
9. dag.wiee.rs/home-made/dstat/
10. github.com/caiopo/pontoon and github.com/caiopo/raft-legacy
11. httpd.apache.org/docs/2.4/programs/ab.html
12. https://kubernetes.io/docs/concepts/services-networking/ingress/

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

This work was partially supported by RNP/CTIC/MCTIC (Brazil) through project ATMOSPHERE and by the Abys project CNPq N^{o} 401364/2014-3.

ORCID

Hylson Netto http://orcid.org/0000-0002-1929-7743
Eduardo Alchieri http://orcid.org/0000-0002-6022-3631

References

[1] Goldberg RP. Architecture of virtual machines. Proceedings of the Workshop on Virtual Computer Systems; Cambridge, MA; 1973. p. 74–112.
[2] Goldberg RP. Survey of virtual machine research. Computer. 1974;7(6):34–45.
[3] Goldberg RP, Mager PS. Virtual machine technology: a bridge from large mainframes to networks of small computers. Compcon Fall 79. Proceedings; Washington, DC; 1979. p. 210–213.
[4] Garfinkel T, Rosenblum M. A virtual machine introspection based architecture for intrusion detection. Proceedings of the Network and Distributed Systems Security Symposium; San Diego, CA; 2003. p. 191–206.
[5] Laureano M, Maziero C, Jamhour E. Intrusion detection in virtual machine environments. Proceedings of 30th Euromicro Conference; Rennes, France; 2004. p. 520–525.
[6] Jiang X, Wang X. "Out-of-the-box" monitoring of vm-based high-interaction honeypots. Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection; Gold Goast, Australia; 2007. p. 198–218.
[7] Júnior VS, Lung LC, Correia M, et al. Intrusion tolerant services through virtualization: a shared memory approach. 24th IEEE International Conference on Advanced Information Networking and Applications; Perth, Australia. IEEE; 2010. p. 768–774.
[8] Dettoni F, Lung LC, Correia M, et al. Byzantine fault-tolerant state machine replication with twin virtual machines. IEEE Symposium on Computers and Communications (ISCC); Split, Croatia; 2013. p. 398–403.
[9] Silva MRX, Lung LC, Magnabosco LQ, et al. Bamcast-byzantine fault-tolerant consensus service for atomic multicast in large-scale networks. 2013 IEEE Symposium on Computers and Communications (ISCC); Split, Croatia. IEEE; 2013. p. 000324–000329.
[10] Wang G, Ng TE. The impact of virtualization on network performance of amazon ec2 data center. INFOCOM, 2010 Proceedings IEEE; San Diego, CA. IEEE; 2010. p. 1–9.
[11] Mell P, Grance T. The nist definition of cloud computing. Gaithersburg (MD): National Institute of Standards & Technology; 2011. SP 800–145.
[12] Soltesz S, Pötzl H, Fiuczynski ME, et al. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. SIGOPS Oper Syst Rev. 2007;41(3):275–287.
[13] Bernstein D. Containers and cloud: from LXC to docker to Kubernetes. IEEE Cloud Comput. 2014;1(3):81–84.
[14] Peinl R, Holzschuher F, Pfitzer F. Docker cluster management for the cloud - survey results and own solution. J Grid Comput. 2016;14:1–18.

[15] Ismail BI, Goortani EM, Ab Karim MB, et al. Evaluation of docker as edge computing platform. 2015 IEEE Conference on Open Systems (ICOS); Melaka, Malaysia. IEEE; 2015. p. 130–135.
[16] Sill A. Emerging standards and organizational patterns in cloud computing. IEEE Cloud Comput. 2015;2(4):72–76.
[17] Verma A, Pedrosa L, Korupolu M, et al. Large-scale cluster management at google with borg. In: European Conference on Computer Systems; Bordeaux, France. New York: ACM; 2015. p. 18.
[18] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm. USENIX Annual Technical Conference; 2014. p. 305–320.
[19] Lamport L. The part-time parliament. ACM Trans Comput Syst. 1998;16(2):133–169.
[20] Schneider FB. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput Surv. 1990;22(4):299–319.
[21] Hadzilacos V, Toueg S. A modular approach to fault-tolerant broadcasts and related problems. Ithaca, NY: Cornell University; 1994.
[22] Ongaro D. Consensus: Bridging theory and practice [dissertation]. Stanford, CA: Stanford University; 2014.
[23] Howard H, Schwarzkopf M, Madhavapeddy A, et al. Raft refloated: do we have consensus? ACM SIGOPS Oper Syst Rev. 2015;49(1):12–21.
[24] Oliveira C, Lung LC, Netto H, et al. Evaluating raft in docker on Kubernetes. In: Świątek J, Tomczak JM, editors. International Conference on Systems Science (ICSS); Wroclaw, Poland. Springer; 2016. p. 123–130. (Advances in Intelligent Systems and Computing; Vol. 539).
[25] Netto HV, Lung LC, Correia M, et al. State machine replication in containers managed by Kubernetes. J Syst Architect. 2017;73:53–59. Special Issue on Reliable Software Technologies for Dependable Distributed Systems.
[26] Mei Y, Liu L, Pu X, et al. Performance analysis of network I/O workloads in virtualized data centers. IEEE Trans Serv Comput. 2013;6(1):48–63.
[27] Ostermann S, Iosup A, Yigitbasi N, et al. A performance analysis of EC2 cloud computing services for scientific computing. Cloud computing; Munich, Germany. Springer; 2009. p. 115–131.
[28] Xavier MG, Neves MV, Rossi FD, et al. Performance evaluation of container-based virtualization for high performance computing environments. 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP); Belfast, UK. IEEE; 2013. p. 233–240.
[29] Amaral M, Polo J, Carrera D, et al. Performance evaluation of microservices architectures using containers. 2015 IEEE 14th International Symposium on Network Computing and Applications; Sep.; Boston, MA; 2015. p. 27–34.
[30] Felter W, Ferreira A, Rajamony R, et al. An updated performance comparison of virtual machines and linux containers. 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS); March; Philadelphia, PA; 2015. p. 171–172.
[31] Kozhirbayev Z, Sinnott RO. A performance comparison of container-based technologies for the cloud. Future Gener Comput Syst. 2017;68:175–182. Available from: http://www.sciencedirect.com/science/article/pii/S0167739X16303041.
[32] Felter W, Ferreira A, Rajamony R, et al. An updated performance comparison of virtual machines and linux containers. International Symposium on Performance Analysis of Systems and Software; Philadelphia, PA. IEEE; 2015. p. 171–172.
[33] Coreos: etcd [https://coreos.com/etcd]; 2018 [cited 2018 Dec 06].
[34] Dwork C, Lynch N, Stockmeyer L. Consensus in the presence of partial synchrony. J ACM. 1988;35(2):288–323.
[35] Ailijiang A, Charapko A, Demirbas M. Consensus in the cloud: Paxos systems demystified. University at Buffalo, The State University of New York; 2016.
[36] Cully B, Lefebvre G, Meyer D, et al. Remus: high availability via asynchronous virtual machine replication. Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation; San Francisco (CA); 2008. p. 161–174.
[37] Bolosky WJ, Bradshaw D, Haagens RB, et al. Paxos replicated state machines as the basis of a high-performance data store. Symposium on Networked Systems Design and Implementation (NSDI); Boston, MA; 2011. p. 141–154.
[38] Marandi PJ, Primi M, Pedone F. High performance state-machine replication. IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN); Hong Kong, China. IEEE; 2011. p. 454–465.
[39] Raft on github [http://raft.github.io]; 2018 [cited 2018 Dec 06].
[40] Gifford DK. Weighted voting for replicated data. Proceedings of the Seventh ACM Symposium on Operating Systems Principles; Pacific Grove, CA. New York, NY: ACM; 1979. p. 150–162.
[41] Felber P, Narasimhan P. Experiences, strategies, and challenges in building fault-tolerant CORBA systems. IEEE Trans Comput. 2004;53(5):497–511.
[42] Bessani A, Sousa J, Alchieri E. State machine replication for the masses with BFT-SMaRt. Proceedings of the International Conference on Dependable Systems and Networks; Atlanta, GA; 2014. p. 355–362.
[43] Shi R, Wang Y. Cheap and available state machine replication. 2016 USENIX Annual Technical Conference (USENIX ATC 16); Denver (CO). USENIX Association; 2016. p. 265–279. Available from: https://www.usenix.org/conference/atc16/technical-sessions/presentation/shi.
[44] Zhao W, Melliar-Smith P, Moser LE. Low latency fault tolerance system. Comput J. 2012;56(6):716–740.
[45] Zhao W, Melliar-Smith PM, Moser LE. Fault tolerance middleware for cloud computing. 2010 IEEE 3rd International Conference on Cloud Computing; July; Miami, FL; 2010. p. 67–74.

[46] Herlihy MP, Wing JM. Linearizability: A correctness condition for concurrent objects. ACM Trans Program Languages Syst. 1990 Jul;12(3):463–492.
[47] Cachin C, Schubert S, Vukolic M. Non-determinism in byzantine fault-tolerant replication. International Conference on Principles of Distributed Systems; Madrid, Spain; 2016. p. 1–16.
[48] Zhao W. Byzantine fault tolerance for nondeterministic applications. Third IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC 2007); Columbia, MD; 2007. p. 108–118.
[49] Zhao W. Performance optimization for state machine replication based on application semantics: a review. J Syst Softw. 2016;112:96–109.
[50] Alchieri E, Dotti F, Mendizabal OM, et al. Reconfiguring parallel state machine replication. Symposium on Reliable Distributed Systems; Hong Kong, China; 2017. p. 104–113.
[51] Alchieri E, Dotti F, Pedone F. Early scheduling in parallel state machine replica. ACM Symposium on Cloud Computing 2018; Carlsbad, CA; 2018. p. 1–14.
[52] Burrows M. The chubby lock service for loosely-coupled distributed systems. Proceedings of the 7th Symposium on Operating Systems Design and Implementation; Seattle, WA; 2006. p. 335–350. (Symposium on Operating Systems Design and Implementation).
[53] Corbett JC, Dean J, Epstein M, et al. Spanner: Google's globally-distributed database. Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation; Berkeley (CA). USENIX Association; 2012. p. 251–264; OSDI'12. Available from: http://dl.acm.org/citation.cfm?id = 2387880.2387905.
[54] Calder B, Wang J, Ogus A, et al. Windows azure storage: a highly available cloud storage service with strong consistency. SOSP; Cascais, Portugal; 2011. p. 143–157.
[55] Glendenning L, Beschastnikh I, Krishnamurthy A, et al. Scalable consistency in scatter. Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles; Cascais, Portugal. New York, NY: ACM; 2011. p. 15–28; SOSP '11. Available from: http://doi.acm.org/10.1145/2043556.2043559.
[56] Hunt P, Konar M, Junqueira FP, et al. Zookeeper: wait-free coordination for internet-scale systems. Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference; Vol. 8; Boston, MA; 2010. p. 11–11.
[57] Vukolić M. The quest for scalable blockchain fabric: proof-of-work vs. BFT replication. In: Camenisch J, Kesdoğan D, editors. Open Problems in Network Security; Cham: Springer International Publishing; 2016. p. 112–125.
[58] Pass R, Shi E. Hybrid consensus: efficient consensus in the permissionless model. IACR Cryptology ePrint Archive. 2016;2016:917.
[59] Eyal I, Gencer AE, Sirer EG, et al. Bitcoin-ng: a scalable blockchain protocol. In: 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16); Santa Clara (CA). USENIX Association; 2016. p. 45–59. Available from: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eyal.
[60] Abraham I, Malkhi D, Nayak K, et al. Solida: a blockchain protocol based on reconfigurable byzantine consensus. Proceedings of the 21st International Conference on Principles of Distributed Systems; Lisboa, Portugal; 2017. p. 25:1–25:19.
[61] Arnautov S, Trach B, Gregor F, et al. Scone: secure linux containers with intel sgx. OSDI; Vol. 16; Savannah, GA; 2016. p. 689–703.
[62] Atomic project: docker and selinux [http://www.projectatomic.io/docs/docker-and-selinux/]; 2018 [citex 2018 Dec 6].
[63] Morabito R, Kjallman J, Komu M. Hypervisors vs. lightweight virtualization: a performance comparison. 2015 IEEE International Conference on Cloud Engineering (IC2E); Tempe, AZ. IEEE; 2015. p. 386–393.
[64] Varma PCV, Chakravarthy KVK, Kumari VV, et al. Analysis of a network IO bottleneck in big data environments based on docker containers. Big Data Res. 2016;3:24–28.