Evaluating Raft in Docker on Kubernetes

Caio Oliveira(✉), Lau Cheuk Lung, Hylson Netto, and Luciana Rech

Universidade Federal de Santa Catarina, Florianopolis, Brazil
caio.po@grad.ufsc.br, {lau.lung,luciana.rech}@ufsc.br,
hylson.vescovi@blumenau.ifc.edu.br

Abstract. In computing systems, some applications require high availability. The creation of copies improves availability, but keeping the copies synchronized requires the replication of the application state. Raft is a consensus algorithm that emerged with an easy understanding logic and a consequently well accepted solution. At infrastructure level, containers offer an alternative for replacing traditional virtual machines in cloud providers. This paper (This project was supported by CNPq proc. 401364/2014-3) evaluates the execution of Raft in physical machines and in Kubernetes, a container management system developed by Google and other companies. Results show similar performance for Raft in both environments.

Keywords: Raft · Performance · Kubernetes · Docker · Containers

1 Introduction

Data centers can manage resources dynamically in an efficient manner with the use of Virtual Machines [12]. This characteristic matches the nature of *cloud computing*, defined by NIST [7] as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (...) that can be rapidly provisioned and released with minimal management effort or service provider interaction". Virtualization at level system, known as Containers [1], provides faster resource allocation, in comparison with virtual machines [3]. Docker is an example of container implementation [11]. Some companies interested in create standards for adopting containers as technology for improvement in resource management founded the *Cloud Native Computing Foundation* (CNCF) [14].

Google has a large experience with containers [16]. Kubernetes is an open source management system for Docker containers [1] which was presented as an initial result from CNCF. Some engineers of Borg [16] (the current container management system at Google) worked in the construction of Kubernetes. Kubernetes replicate containers with the aim of improving availability. Failed containers are recreated by Kubernetes, but the state of the application inside the container is not restored. External volumes can persist the state. However, the volumes should be protected against failures and concurrent access to the volume have to be controlled by the application.

Raft [10] is an algorithm derived from Paxos [6]. It can be used to implement replicated state machines (SMR) [13] in local area networks (LAN). Raft emerged as a understandable algorithm, when compared to Paxos. Consequently, many Raft implementations became available[1] in various programming languages. With Raft, all requests send to any replicated container will be executed on all replicas, in the same order. Raft can be applied in Kubernetes at application level, i.e., inside containers. There are some evaluations of Raft [5,9] but its characterization of terms of latency and throughput are still incipient.

This paper brought an evaluation of Raft to provide state machine replication. We compare its execution in containers implemented by Docker on the Kubernetes environment with the execution directly in bare metal (i.e. physical machines). Although throughput is quite different and higher in physical machines, we found that clients observe similar latencies in both environments. Kubernetes provides many management features that could compensate the overhead perceived in the throughput measurements.

The remainder of this paper is organized as follows. Section 2 presents concepts about containers in Kubernetes. In Sect. 3 we take an overview in Raft. Section 4 brings the adaptions make to run Raft in Kubernetes. In Sect. 5 we evaluated the executions of Raft and finally in Sect. 6 we conclude the paper.

2 Containers and Kubernetes

Virtualization at system level appeared in the FreeBSD operational system as an extended version of the *chroot* command called *jail* [1]. Next, Solaris improved this resource, which was called *zone*. Containers are instantiated from static images. The state of a container can be defined by all data stored inside the container since its instantiation from the image. When a container is destroyed, its state is lost. It is not common to maintain session data inside containers. Thus, containers can be considered stateless virtual machines. Images of containers are usually small because they use a layered file system, e.g. `aufs`. That way, only files that do not exist in its host are effectively stored in the container image. With a level system virtualization and a layered file system, containers can be created and destroyed faster than traditional virtual machines [3]. It brings a more dynamic resource management capability to data centers. An example of container implementation is Docker [11].

Google created Kubernetes [1] as an evolution of its current management system called Borg [16]. Kubernetes has many features originally designed in Borg. For example, considers a web server and its logs, which are consumed by a log analyzer. Host these applications in different containers is recommended because it benefits the individual software maintenance. There containers should remain on the same machine, to avoid the effort of sending the log over the network. A feature of Borg called *Alloc* maintains containers together in the same machine. Kubernetes has a component called *POD*, which also maintains containers together on the same machine.

[1] raft.github.io.

Kubernetes is composed of machines (virtual or physical) called nodes (Fig. 1). The component *POD* contain one or more containers. Each POD receives a network address. Containers inside a POD can share resources such as external data volumes. A *firewall* forwards requests from clients to nodes in the Kubernetes cluster. Each request will be delivered to only one node. The load balance policy is defined by the rules of the firewall, which is a component external to the Kubernetes cluster. The *proxy* component forwards requests to PODs, which can be replicated or not. When PODs are replicated, requests are distributed in a round-robin fashion. PODs are managed by the *kubelet* component. The *kubelet* also sends data about monitoring of containers to the main node.

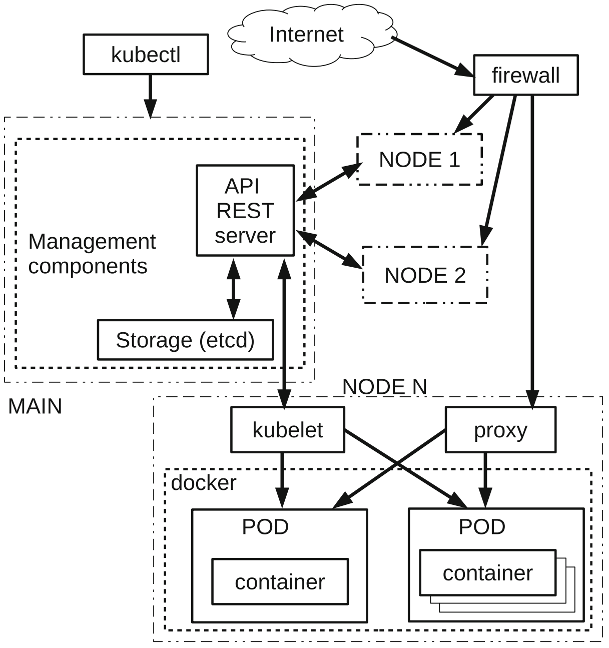


Fig. 1. Kubernetes architecture.

The *main* node of Kubernetes contains the management components. All information about the Kubernetes cluster is persisted in a storage component. The tool *etcd* [2] implements the storage in Kubernetes. Components interact via *REST* APIs. Components use the API Server to save and retrieve information. A human operation can interact with the cluster using the *kubectl* command interface. Some examples of commands are the creation of PODs, checking the health of the cluster and getting the description of PODs that are running.

3 Raft

Raft [10] is a consensus protocol designed to be of easy understanding, in comparison with Paxos [6] which is one of the most known consensus protocol. As consequence of the effort to create a simple protocol, many implementations of Raft are available [4]. Raft is already available in the *etcd* component[2] and can distribute the storage across the network, making it fault tolerant [15]. A set of replicas that have to maintain the same state is called a *configuration*. Raft ensure that replicas members of a configuration execute requests in the same order, what enables the maintenance of a replicated state on the replicas [13]. The system still works even if a minority of replica fail by crash. Formally, the system must have $n = 2f + 1$ replicas in the system, on which up to f replicas can fail. In Raft, crash failures are tolerated.

Replicas can act as leader, follower or candidate. An unique replica acts as a leader, determining the order of requests. Clients interact only with the leader. If a client interact with a follower replica, the client will be informed about the address of the leader replica. Follower replicas obey the proposal of the leader, running the requests in the established order. When a leader fails, elections are started and replicas can become candidate. The most updated replica in the system wins the election and becomes the new leader, sending heartbeat messages to other replicas to establish its authority and prevent new elections.

Each replica has to receive periodically answers from the leader, through a heartbeat mechanism. If leader does not answer after a timeout, replicas become candidate and start elections. Replicas use randomized timeouts (in a fixed interval, e.g., 150–300 ms) to prevent split votes. Initially, a candidate replica sends `RequestVote` messages to all replicas, which answer by `Vote` messages. A replica only does not accept a leader if this replica has an state more updated than the state of the proposer, and when a new leader was not elected yet. A new leader is elected when it is accepted by a majority of replicas.

An elected leader receives requests, assigns order to them and send a list of ordered requests to all replicas. The message which contains this list is called `AppendEntry`. On receiving this list, a follower replica updates its own list and answers the leader about this update. When the leader acknowledges that a majority of replicas (including itself) have new updated requests, the leader executes these new requests, sends answers to the clients (replies) and sends to replicas a list with the executed requests. The follower replicas also execute these requests and update their states. In Raft site [4] an animation allows the simulation of some actions (e.g. faults, request sending) in a quorum of five servers. In scenario the system can tolerate two faults, because the crash fault tolerance rule ($n = 2 * f + 1$) requires $n = 5$ servers to tolerate $f = 2$ faults.

[2] github.com/coreos/etcd/tree/master/raft.

4 Modifying Raft for Kubernetes

We choose one[3] of the many available implementations of Raft [4]. We choose an implementation in *Go*, the language on which Kubernetes was developed. Some modifications were done to allow the execution of Raft in the Kubernetes environment. The source code is available in GitHub[4]. We modified the original chosen implementation about the following characteristics:

– *Replica discovery.* Replicas communicate to each other via the IP address. When a replica enters in the system, it have to know the IP of the other replicas. To get this addresses, each replica was provided with a call to the *endpoints*[5] Kubernetes API, which returns a list with the IP of all replicas. This action is necessary because each container receives a dynamic IP during its instantiation. The API call has the format of an URL like `http://ip-of-main-node:port-of-api-server/api/v1/endpoints`. Usually, the port of the API server is the 8080 port. Replicas query the *endpoints* API periodically to keep updated with new replicas that entered in the system. These modifications are in the `caiopo/raft` repository, inside the `raft.go` program. This program also contains a hard-coded IP prefix like "18.16." (this IP prefix was defined inside the Flannel configuration (see Sect. 5.1). The *endpoints* API returns the IP of all containers. So, we compared the prefix with each returned address and consider only the IP of the our Raft replicated containers. It is possible to specify metadata in the creation of the container and get endpoints of an specific tag, but we did use this feature on our container.
– *Accepting of requests by any replica.* In Raft, only the leader can receive requests from clients. When a non-leader replica receives requests, they are ignored and the client is informed with the address of the leader. We modify Raft to allow each follower replica answer to clients about requests that they receive. It is mandatory because Kubernetes does an internal load balance (via `proxy` component, Sect. 2), delivering requests to any replicated container. This is done with a direct verification like "`if t.node.State == Leader`", being the block of code inside the *else* clause responsible by sending the requests to the leader and waiting for the answer. Modifications of this default behavior are in the `caiopo/pontoon` repository, inside the `http_transport.go` program. Another modifications in this program included the addition of commands like `hash`, which was used to get the hash over the log of executed commands in the container. We could compare the hash of all replicas in order to check if the state remains equals in all replicas after the execution of each test.

The modified source code of Raft is installed in a Docker container and is available in Docker Hub[6] under the tag `caiopo/raft`.

[3] github.com/mreiferson/pontoon.
[4] github.com/caiopo/pontoon and github.com/caiopo/raft.
[5] kubernetes.io/docs/api-reference/v1/definitions/#_v1_endpoints.
[6] hub.docker.com.

5 Evaluation

In this section we evaluate the performance of Raft in physical machines and in Kubernetes. Containers are virtual machines at system level. Therefore, the system will provide similar performance when running Raft in physical machines and in Docker containers on Kubernetes [3].

5.1 Experiment

An experiment was conducted to investigate the presented conjectures. We used a cluster with four computers Intel i7 3.5 GHz, QuadCore, cache L3 8 MB, 12 GB RAM, 1TB HD 7200 RPM. Machines are connected through an Ethernet 10/100 MBits network. Ubuntu Server 14.04.3 64 bits with kernel 3.19.0-42 was installed in the computers. The Kubernetes 1.1.7 was used. Docker containers communicate among them between physical machines via Flannel virtual network[7].

Raft was executed in Kubernetes and directly on the physical machines, using 3 replicas. The number of clients that simultaneously accessed Raft was varied following the sequence 4, 8, 16, 32 and 64. When using only Kubernetes, the number of replicas was three. To simulate the access of clients to Raft, the Apache HTTP server benchmarking tool[8] was used.

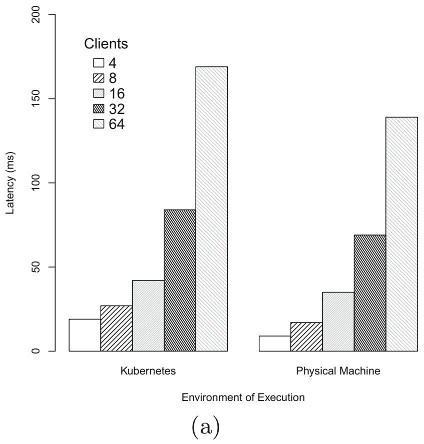
5.2 Results and Discussion

The evaluated latencies presented stable results, with low standard deviation (except for 4 and 8 clients), as presents Table 1. The performance of Raft is faster when running in physical machines. Although the overhead of Kubernetes starts high (111.1 %) for few clients, it decreases as more clients start to make requests, stabilizing around 21 % for 16 or more simultaneous access. Also represented in Fig. 2, latencies increase as more clients enter in the system.

Table 1. Measurements of Raft

Cli.	Req.	Kubernetes			Physical machine			Ku/Phy (%)
		Latency (ms)	St.Dev. (ms)	Time of test (s)	Latency (ms)	St.Dev. (ms)	Time of test (s)	
4	16000	19	62.8	75.3	9	1.0	34.5	111.1
8	12000	27	57.1	40.2	17	1.4	26.2	58.8
16	8000	42	2.3	12.1	35	2.1	17.3	20.0
32	8000	84	4.1	21.0	69	3.3	17.3	21.7
64	8000	169	9	21.2	139	7.7	17.5	21.6

[7] coreos.com/flannel.
[8] httpd.apache.org/docs/2.4/programs/ab.html.



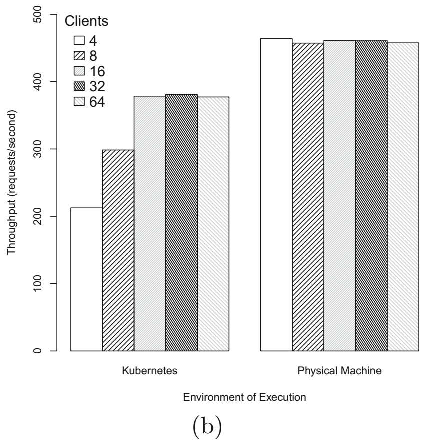


Fig. 2. (a) Latency and (b) Throughput of Raft in the evaluated environments.

Throughput of Raft increases as more clients access Raft in Kubernetes. For more than 16 clients, it remains around 380 requests per second. When Raft runs in physical machines, throughput is stable and presents the better value among all measurements. However, it does not scale as more clients send requests. This superior cut point can occurs because of a limitation in our chosen Raft implementation, which does not implement batch of messages.

Ongaro [9] have preliminary tests of performance in which throughput scales as more replicas enter in the system, but latency is evaluated considering only one client accessing the system. Another work [5] repeat the Raft authors' performance analysis, not considering latency perceived by clients nor throughput. Although executing Raft in Kubernetes presents throughput lower than when running in physical machines, we argue that the reduction of approximately 17.4 % is compensated by the benefits offered by the Kubernetes features.

6 Conclusion

This paper presents an evaluation of the Raft algorithm running on physical machines and in Docker containers managed by Kubernetes. We modified Raft to be capable of dynamically discover replicas and to allow clients send requests to any replica of Raft (i.e., not only the leader). Results show that Kubernetes provides competitive throughput when compared with the execution in bare metal. As more clients enter in the system, the latency perceived by the clients is similar in both environments.

Improvements in this work includes the usage of other consensus algorithms. For example, EPaxos [8] did not use leader, which could make better use of the load balance provided by Kubernetes. A more robust implementation of Raft can be used in order to achieve more efficient results. With containers, we can easily increase the number of replicas, when comparing with the context of traditional virtual machines. New experiments could consider scenarios with more faults. Applications that require high availability but are exposed to many risks (like sites exposed on Internet) are expected to suffer more faults than applications executed in local area networks or inside organizations.

References

1. Bernstein, D.: Containers and cloud: from lxc to docker to kuber-netes. IEEE Cloud Comput. **1**(3), 81–84 (2014)
2. CoreOS. etcd (2016). https://coreos.com/etcd. Accessed 12 May 2016
3. Felter, W., et al.: An updated performance comparison of virtual machines, Linux containers. In: International Symposium on Performance Analysis of Systems and Software, pp. 171–172. IEEE (2015)
4. GitHub. Raft (2016). http://raft.github.io. Accessed 12 May 2016
5. Howard, H., et al.: Raft refloated: do we have consensus? ACM SIGOPS Oper. Syst. Rev. **49**(1), 12–21 (2015)
6. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
7. Mell, P., Grance, T.: The NIST definition of cloud computing. Technical report Spp. 800–145. Gaithersburg, MD, United States: National Institute of Standards & Technology (2011)
8. Moraru, I., Andersen, D.G., Kaminsky, M.: There is more consensus in egalitarian parliaments. In: Proceedings of the Twenty-Fourth Symposium on Operating Systems Principles, pp. 358–372. ACM (2013)
9. Ongaro, D.: Consensus: bridging theory and practice. Ph.D. thesis. Stanford University (2014)
10. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: USENIX Annual Technical Conference, pp. 305–320 (2014)
11. Peinl, R., Holzschuher, F., Pfitzer, F.: Docker cluster management for the cloud - survey results, own solution. J. Grid Comput. **14**, 1–18 (2016)
12. Popek, G.J., Goldberg, R.P.: Formal requirements for virtualizable third generation architectures. Commun. ACM **17**(7), 412–421 (1974). ISSN:0001–0782
13. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. **22**(4), 299–319 (1990)
14. Sill, A.: Emerging standards, organizational Patterns in cloud computing. IEEE Cloud Comput. **2**(4), 72–76 (2015). ISSN:2325–6095
15. Toffetti, G., et al.: An architecture for self-managing microservices. In: Proceedings of the 1st International Workshop on Automated Incident Management in Cloud, pp. 19–24. ACM (2015)
16. Verma, A., et al.: Large-scale cluster management at Google with Borg. In: European Conference on Computer Systems, p. 18. ACM (2015)