Koordinator: A Service Approach for Replicating Docker Containers in Kubernetes

Hylson Vescovi Netto*, Aldelir Fernando Luiz*, Miguel Correia†, Luciana de Oliveira Rech‡, Caio Pereira Oliveira‡
*College of Blumenau, Federal Institute Catarinense - Brazil
†INESC-ID, Instituto Superior Técnico, Universidade de Lisboa - Portugal
‡Department of Informatics and Statistics, Federal University of Santa Catarina - Brazil
{hylson.vescovi, aldelir.luiz}@ifc.edu.br, miguel.p.correia@tecnico.ulisboa.pt, luciana.rech@ufsc.br, caiopoliveira@gmail.com

***Abstract*—Container-based virtualization technologies such as Docker and Kubernetes are being adopted by cloud service providers due to their simpler deployment, better performance, and lower memory footprint in relation to hypervisor-based virtualization. Kubernetes supports basic replication for availability, but does not provide strong consistency and may corrupt application state in case there is a fault. This paper presents a state machine replication scheme for Kubernetes that provides high availability and integrity with strong consistency. Replica coordination is provided as a service, with lightweight coupling to applications. Experimental results show the solution feasibility.**

I. Introduction

The increasing adoption of cloud computing [1] by organizations and individuals is driving the use of *virtualization* technology [2], [3]. Virtualization is extremely important in cloud computing mainly to isolate the consumers and to control the provisioning of resources.

The prevalent virtualization technology in cloud data centers is based on *hypervisors*. Each server has an hypervisor (e.g., Xen or KVM) on top of which are executed *virtual machines* (VMs) containing their own operating system (OS) and user software. However, *container-based virtualization* (or OS-level virtualization) is gaining increasing acceptance [4]. In this form of virtualization, several containers are executed on top of the same OS, with benefits such as simpler deployment, better performance, and lower memory footprint, in comparison to hypervisor-based virtualization.

Docker is the most popular container-based virtualization technology [5]. Docker supports the execution of containers on top of Linux. However, it does not support management or orchestration of containers in a cluster environment. Therefore, engineers at Google developed Borg [6] and, the better known, Kubernetes [7], a system that allows controlling the lifecycle of containers on a cluster environment.

Kubernetes supports basic application *replication* for availability. Specifically, it allows defining the number of container replicas that should be running and, whenever a replica stops, it starts a new one from an image. However, this scheme does not provide strong consistency. For instance, if a replica fails two problems may happen: part of the application state may be lost (in case it is stored in the containers) or the state may become corrupted (in case it is stored in a backend data store, outside of the containers). In relation to the latter issue, if there is a backend data store, access concurrency must be managed to prevent corruption.

This paper presents *Koordinator*, a new container replica coordination approach that provides availability and integrity with strong consistency in Kubernetes. Koordinator is based on state machine replication (SMR) [8], an approach that keeps replicas consistent even if some of them fail, providing high availability and integrity. Koordinator is provided *as a service*, i.e., on top of Kubernetes, with lightweight coupling with the application being replicated.

The rest of this paper is organized as follows. Section II briefly introduces container-based virtualization. Section III presents Koordinator. The experimental validation and results are presented in Section IV. Section V discusses related works and Section VI concludes the paper.

II. Container-Based Virtualization

Container-based virtualization, or OS-level virtualization, is implemented by an OS, as the second name suggests (Figure 1). Each *container* contains libraries and application code. Isolation between containers is provided by the OS itself, e.g., by mechanisms such as Linux *cgroups* and *namespaces* in Docker and Linux Containers (LXC). Container management software provides functionality such as the creation of containers.

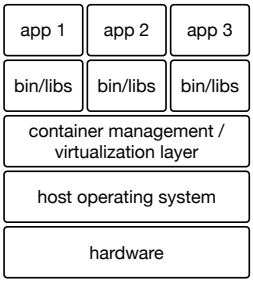


Fig. 1: Container-based Virtualization.

In large-scale systems, such as cloud data centers, there may be a vast number of containers, so their management and orchestration with basic tools to create/delete containers may be complicated. Kubernetes is a container management system that simplifies such management in a cluster, i.e., in a group of physical or virtual machines. The two major alternatives to Kubernetes are Docker Swarm and Apache Mesos.

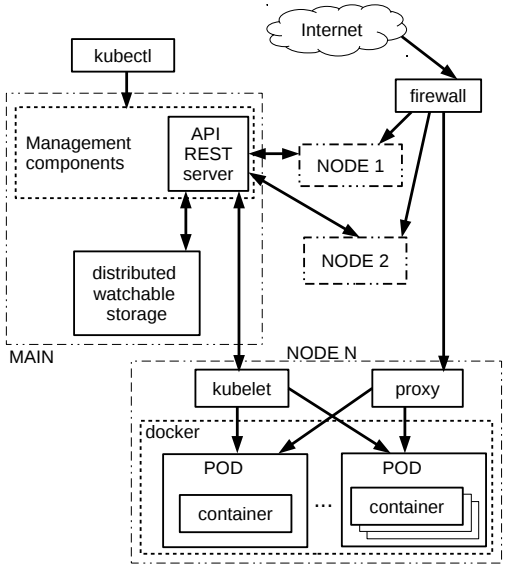


Fig. 2: Architecture of Kubernetes.

Figure 2 presents the Kubernetes architecture. Each machine, either virtual or physical, is designated by *node*. A *pod* is a group of one or more containers that must be in the same machine and constitutes the minimal management unit. Pods receive network address and are allocated to nodes. Containers inside a pod share resources, such as volumes where they can write and read data. Each node contains an agent called *kubelet* that manages its pods, containers, images, and other resources.

A Kubernetes cluster often provides services that process *requests* incoming from clients (e.g., REST/HTTP requests). A typical interaction is the following. A client request reaches the cluster through its firewall (top-right of the figure), which forwards it to the proxy at a node. That proxy forwards the request to the kubelet. The kubelet does load balancing, in case the service is replicated, and forwards the request to a container in a pod that processes it.

The node where Kubernetes' management components run is denominated *main node*. Some important management components are the *etcd* service that does notification (e.g., notifies certain components when data is created or updated) and *kubectl* that is used by operators to interact with the Kubernetes cluster. Most communication between components uses the REST API.

III. KOORDINATOR

This section presents our solution for replicating stateful containers in Kubernetes.

A. Approach

The management of stateful containers in Kubernetes depends on the number of containers replicas being executed. For single-instance, non-replicated, containers, the state has to be maintained securely by connecting the container to persistent storage, e.g., a disk volume[1]. In that situation, if the container fails, the availability of the application is compromised. Replicating the container can improve the application availability (the failure of a replica does not impair the application availability) and provide better throughput (because requests are distributed among replicas via the load balancing scheme provided by Kubernetes). However, when multiple containers access shared data in a persistent volume, this access has to be coordinated, so that concurrent requests are properly executed.

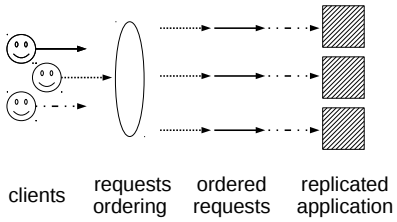


Fig. 3: Ordering requests for simultaneous clients accessing a replicated stateful application.

A well-known approach to coordinate request execution in a replicated application is state machine replication (SMR) [8]. With SMR, concurrent requests are ordered by a distributed consensus algorithm and their execution happens in the same way in all replicas (Figure 3). Paxos, Raft, PBFT, MinBFT, and BFT-Smart are examples of such consensus algorithms [9], [10], [11], [12], [13]. However, using such schemes with containers is far from trivial. Although there are SMR libraries available, e.g., Raft [10] and BFT-Smart [13], applications have to be modified to use such software. The interface of an SMR implementation can be complex to integrate with already existing applications [14].

A possible strategy to coordinate replicated applications is *incorporating coordination in the environment where the applications are executed*. At least three possible approaches can be considered when boarding a consensus algorithm in an existing platform: integration, interception, and service [15], [16]. *Integration* means building or modifying a component in a system with the aim of adding some feature. There has been a previous effort to integrate coordination in Kubernetes with a new consensus protocol [17]. It this paper, instead, we want to use available *crash fault-tolerant* consensus libraries which are stable and continuously being improved, such as BFT-Smart[2] and Raft. The *interception* approach requires that messages sent to the receiver are captured and passed to the communication layer. A practical example of this approach is the use of operating system interfaces to intercept system calls related to the application [18]. Both integration and interception are transparent for the clients which are accessing the replicated system.

In contrast with the other approaches, the *service approach* – the one we adopt – incorporates new functionality in the system, in the form of a service layer on top of that system, not as an integral part of it [19], [20]. In our case, this layer is inserted between the client and the replicated containers to

[1]https://kubernetes.io/docs/tasks/run-application/run-single-instance-stateful-application/

[2]BFT-Smart is better known for tolerating Byzantine faults, but is can also be configured to tolerate only crash faults, with improved performance.

coordinate the requests sent to the application, which is inside those containers.

B. Coordination as as Service

Our proposal to incorporate coordination as a service in Kubernetes resulted in a system we call *Koordinator*. Koordinator provides a service that allows a set of clients to coordinate application state updates (*writes*) in containers on Kubernetes. Koordinator is a container layer that orders the requests, solving consensus (Figure 4a). State *reads* from the application can be done directly from any of the containers, bypassing the coordination layer, allowing these operations to be done efficiently (Figure 4b). The write and read operations are detailed next.

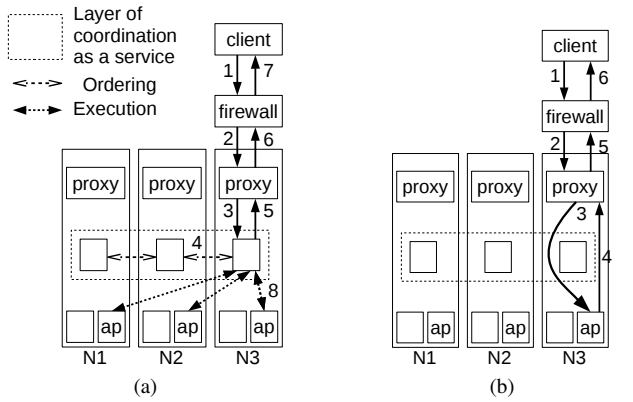


Fig. 4: Koordinator: (a) coordinating updates and (b) direct reading in the replicated application.

Clients send write requests to the cluster firewall (Figure 4a, step 1), which forwards them to one of the nodes in the cluster (N3). Each request is received by the proxy (step 2) and delivered to the coordination layer (step 3). Koordinator executes a distributed consensus algorithm to define the order of requests (step 4), answering to the client when the consensus is achieved (steps 5, 6, and 7). Koordinator proceeds to the execution of requests in the order they were assigned by the consensus algorithm (step 8). In Koordinator, only leader-based consensus algorithms are considered. That way, the communication between the ordering and execution layers is done through the consensus leader and the application replicas.

When reading the state of the application, client requests are delivered directly to one of the application replicas (steps 1, 2 and 3). After processing the request, the application answer is sent to the client (steps 4, 5 and 6).

The read protocol is extremely important performance-wise, as most workloads have much more reads than writes [21] and the read protocol is much lighter than the write protocol. However, the read protocol also implies that an application replicated using this scheme provides only eventual consistency [22]. With this consistency it is ensured that after an application state update (write) without further updates, all reader clients will get the new updated value. However, during the execution of update requests, clients that are doing reads can receive different replies (either the old or the new one). This inconsistency window is defined by factors such as network latency, request time processing, and the number of application replicas. Social networks are common examples of applications that provide only eventual consistency.

Submitting reading requests through Koordinator (as in Figure 4a) could provide strong consistency by ordering them. However, in this paper we provide strong consistency by creating only one instance of the application. Strong consistency is required, for example, in collaborative document edition.

Algorithm 1 Code executed by replica p_i

{ **Code executed if replica is the consensus coordinator** }

Shared Variables and Data Structures:

1: $replies\langle\rangle = \emptyset$ {Map of replies from replicas}
2: $ordered\_request\_buffer\langle\rangle = \emptyset$ {A hash table to store ordered requests}
3: $last\_stable\_request = \perp$ {The last completed/answered request}
4: $agreement\_replicas = \emptyset$ {Set of Raft replicas}
5: $application\_replicas = \emptyset$ {Set of Application replicas}

Main Task:

6: **foreach** $r_i \in application\_replicas$ **do**
7: `start` $Execution\_Task(r_i, i)$ {i is the i-th process id}

Execution Task:

8: $\langle r_i, p_i\rangle \leftarrow \{r_i, i\}$ {Setting received arguments/parameters}
9: $position \leftarrow 1$ {Position of the next request}
10: **repeat** $forever$
11: **if** $|request\_queue| > 0$ **then**
12: $operation \leftarrow ordered\_request\_buffer\langle position\rangle$
13: $send(p_i, \langle$`ORDERED-REQUEST`$, operation\rangle)$ to r_i
14: **wait until** $(receive(\langle$`REPLY`$, result\rangle)$ from $r_i) \vee (\Delta_{r_i}$ expired)
15: **if** Δ_{r_i} **not** expired **then**
16: $response \leftarrow \{result$ **from** message $\langle$`REPLY`$, result\rangle\}$
17: $replies\langle position\rangle \leftarrow replies\langle position\rangle \cup \{\langle p_i, response\rangle\}$
18: $majority \leftarrow \lfloor |application\_replicas|/2\rfloor + 1$
19: $position \leftarrow position + 1$
20: **if** $(|replies\langle position\rangle| \geq majority) \wedge (last\_stable\_request < position)$ **then**
21: $last\_stable\_request \leftarrow position$

Listener Task:

22: **upon** $receive(\langle$`WRITE-REQUEST`$, operation\rangle)$ from client c_j
23: $\langle status, order\rangle \leftarrow$ `underlying_consensus`$(operation)$ {Ordering request through a consensus algorithm}
24: $ordered\_request\_buffer\langle order\rangle \leftarrow operation$
25: $send(p_i, \langle$`REPLY`$, status\rangle)$ to c_j

{ **Code executed if the replica is not the coordinator** }

Listener Tasks:

26: **upon** $receive(\langle$`READ-REQUEST`$, operation\rangle)$ from client c_j
27: $command \leftarrow \{operation$ **from** message $\langle$`READ-REQUEST`$, operation\rangle\}$
28: $result \leftarrow$`execute`$(command)$
29: $send(p_i, \langle$`REPLY`$, result\rangle)$ to c_j
30: **upon** $receive(\langle$`ORDERED-REQUEST`$, operation\rangle)$ from process p_j
31: $command \leftarrow \{operation$ **from** message $\langle$`ORDERED-REQUEST`$, operation\rangle\}$
32: $result \leftarrow$`execute`$(command)$
33: $send(p_i, \langle$`REPLY`$, result\rangle)$ to p_j

C. System Model

Before presenting the Koordinator's algorithm in a more rigorous way, we present the system model.

There is a set $\mathcal{C}$ of replicated containers, out of which at most of $f < |\mathcal{C}|/2$ can suffer crash faults. Note that containers do not recover from failures, instead new containers can be launched to replace faulty ones, re-configuring the system.

We assume an eventually synchronous system, i.e., that communication delays eventually stabilize [23]. Clients and servers are connected through reliable channels [24]. This implies that messages sent will be eventually received.

To order incoming requests, we consider that our system uses Raft as consensus algorithm. In this sense, as soon as a request is ordered, the protocol notifies the client using an ACK. Ordered requests enter in an execution queue. Note that as our coordination layer is a service, we could use another consensus algorithm like Paxos [14] or BFT-Smart [13].

D. Detailed Algorithm

In the following, we present details of the protocol operation. Specifically, Algorithm 1 describes the behavior of the replicas, distinguishing the replica that coordinates the consensus (the leader) from the rest. The algorithm has four tasks that execute in isolation and some variables to keep up consistency of the replica state (lines 1 – 5).

The code of lines 1 to 21 refers to the coordinator replica, which is the coordinator of Raft. Only coordinator replica initialize variables in lines 1 to 5 and runs execution tasks, one for each application replica. These tasks manage communication between each node of the application layer and the coordination layer. The remaining replicas execute these tasks only if they eventually assume the leader role.

When the coordinator replica starts, it first executes the *Main Task* that creates and starts an execution task for each application replica in the set $application\_replicas$ (lines 6 and 7). r_i and i are parameters used in the tasks. As we said in Section III-B, we distinguish between two operation types: *read*, an operation that reads data, and *write*, an operation that creates or updates data. As explained before, our protocol handles only write requests, read requests are handled directly to the application layer bypassing protocol (lines 26 – 29).

To execute a write operation, the client sends a message $\langle$`WRITE-REQUEST`, $operation\rangle$ to the coordinator replica. The *Listener Task* of the coordinator handles these requests (lines 22 – 25). Upon receiving a write operation – in line 23 –, the coordinator first sends that operation to the underlying consensus protocol that defines its order. After the order is defined, the coordinator stores the operation in a buffer that will be consumed by the execution tasks in order (line 24). At last, the coordinator replies to the client that requested the operation (line 25). Note that the operation will be processed later, as soon as it consumed by the execution tasks (line 12).

Next, task i enters in an infinite loop to process requests (lines 10 – 21). When task i retrieves an ordered operation, it immediately sends it to the i-th application replica and waits for its reply (lines 12 – 14). If it received the reply, so the timeout has not expired, the task gets the result of the operation and stores it in a response map (e.g., replies data structure). This happens in lines 16 and 17.

In next steps, (i) the task determines a majority value based on the number of active (application) replicas (up to $\lceil N/2 \rceil - 1$ application replicas can crash, leaving the system, line 18); and (ii) get ready for the next request (line 19). Finally, if there is a reply with majority – which states that a reasonable amount of replicas executed that request –, this request number indicates the last stable request, or the last answered request (lines 20, 21). Note that, if the timeout expires, the queue position will remain the same. In that case, in next loop the tasks will try to execute this request again.

IV. Evaluation and Results

We implemented a prototype of Koordinator in Go (available at github.com/caiopo/raft) and evaluated it experimentally. The environment consisted of a set of four computers, each one with a Quad-Core Intel i7 processor working at 3.5 GHz, 12GB of memory, 1TB of disk with 7200RPM, and one 10/100 Mbits fast ethernet card. Each node run Ubuntu 14.04.3 64 bits, kernel 3.19.0-42.

For the experiments we used Kubernetes 1.1.7. We distributed four machines as following: one machine ran the main node, and the rest ran as container execution nodes. We used Docker containers. Raft was instantiated as consensus protocol in the coordination layer.

We used a microbenchmark to evaluate some scenarios of our proposal. The microbenchmark ran an application based on a text repository, called logger (available at hub.docker.com/r/caiopo/logger). The experiments involved three scenarios:

1) *Only writers* (e1). The system is evaluated with clients doing only writes. An example of this scenario is an application that registers operations in a log.
2) *Reading with strong consistency* (e2). To enable clients to read data with strong consistency, this scenario has a single instance of the application container.
3) *Reading with eventual consistency* (e3). Clients can read data faster if only eventual consistency is guaranteed. In this case, the application can be replicated to allow a higher read throughput and lower latency. Moreover, additional replicas provide fault tolerance. In this scenario we have three replicas of the application.

Clients (writer or reader) are located in the main node (Figure 5). We created a firewall (FW_w) to direct all writing operations to the Raft leader replica. The firewall is inside a container, receiving requests via proxy. The firewall and the Raft leader are on the same node, to where all writers send requests. When clients are reading data from the application replicas, Kubernetes provides internal round-robin distribution of the requests via its proxy (e3 scenario). During each experiment execution, the Raft leader is instantiated in a different node, according to the elections (Table I). Furthermore, the node on which readings are done is different from the node that hosts the leader, to avoid overloading the node.

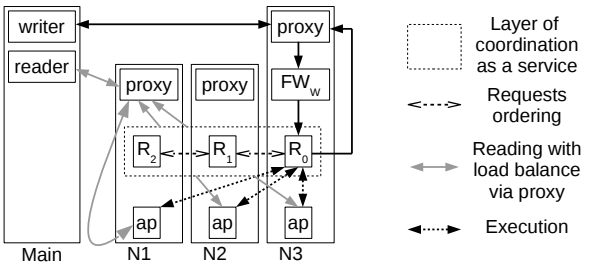


Fig. 5: Configuration used in the experiments.

TABLE I: Nodes with Raft leader and targeting readings

Experiment	Raft leader	Readings to
e1	N1	-
e2	N2	N3
e3	N2	N1

TABLE II: Experimental results

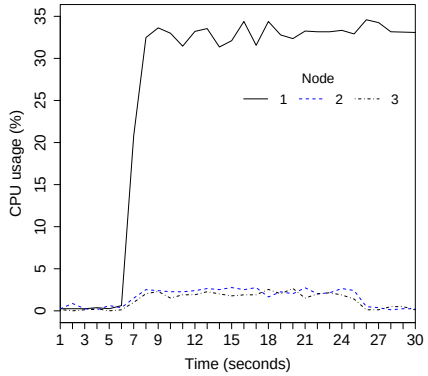
	writers (16)			readers (256)		
	latency (ms)			latency (ms)		
exp.	mean	st.dev.	req/s	mean	st.dev.	req/s
e1	37	1.8	431	-	-	-
e2	38	13.5	420	5	27.1	50318
e3	44	43.9	363	10	32.4	24835

The workload is the following. 16 writers simultaneously send 8000 requests and 256 readers simultaneously send 80,000 requests. The Apache benchmarking tool (ab) is used to create and send the requests. The payload is around 100 bytes for requests and 5 bytes for replies. Each client only sends a new request upon receiving the reply for the previous request.

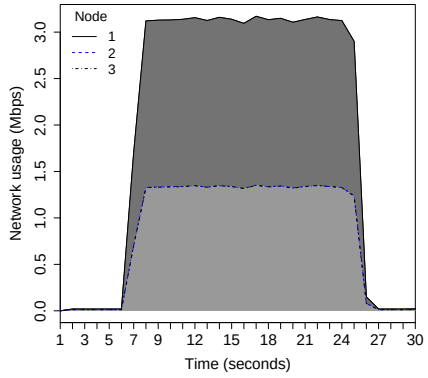
Clouds usually offers a pay-as-you-go cost model. That way, we measured the resource consumption of CPU and network of the machines used in the experiment. The dstat tool was used for monitoring the resources during the experiments execution (available at dag.wiee.rs/home-made/dstat/).

The main results for the three sets of experiments are shown in Table II. The latency of writing clients is barely affected by the inclusion of reading clients, as observed when comparing the first two rows of results (e1 and e2). In the case of only one application instance (e2), requests are all executed by the node which receives them. With more application replicas (e3), requests are sent to other nodes, via proxy load balancing, increasing latency because of the network usage (5ms to 10ms). Network impacts also in the throughput (request per second), from 50 hundred in e2 to 25 hundred in e3.

We show the resource consumption (CPU and network) for the two most representative scenarios: with and without readers (e1 and e2). In the first scenario (e1) the node with the Raft leader (N1) consumes more resources than other nodes because only writing operations are done in the leader (Figure 6). Nodes 2 and 3 show the same network behavior (Figure 6b). When reading data from one application instance (e2), the node which hosts the application container presents a peak of usage (Figure 7a). However the main work remains to the leader, hosted in N2. The Raft leader consumes more network bandwidth, except when readings happen (Figure 7b).



(a)



(b)

Fig. 6: CPU (a) and network (b) usage in experiment e1.

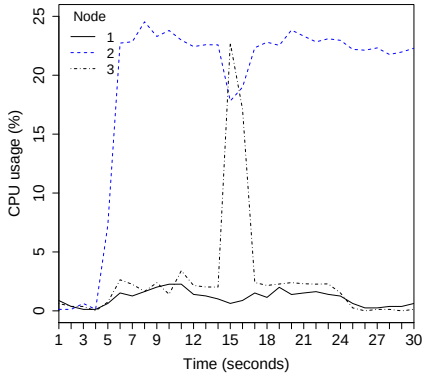
V. Related Work

SMR is a problem known since the 1980s [8]. Lamport presented Paxos, which is one of the algorithms that most inspired work in the area [9]. The first efficient Byzantine fault-tolerant SMR algorithm was created by Castro and Liskov and is known as PBFT [11]. Veronese et al. designed a set of SMR algorithms that use a trusted component to be efficient in performance and number of replicas [12].

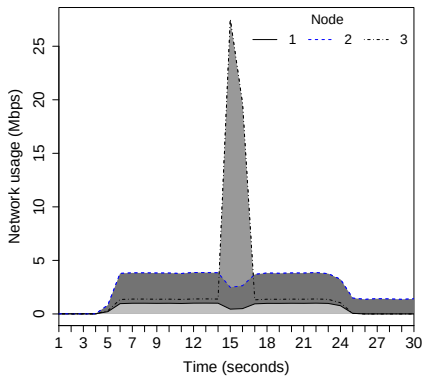
Currently there are just a few stable SMR library implementations, as these algorithms are hard to implement. An interesting one is BFT-Smart [13] that provides advanced functions like state transfer and reconfiguration for dynamic environments. Evaluation of BFT-Smart in containers has shown higher performance comparing with traditional VMs [25]. There are also a few practical implementations of the Raft algorithm, which tolerates only crashes [10]. Prime is another stable Byzantine fault-tolerant SMR library publicly available [26].

A recent work from some of the authors integrated SMR in Kubernetes using Raft [17]. The goal was to provide transparent coordination and reduce the size of application containers. Raft integrated in Kubernetes presented performance 17.4% worse than Raft running directly in a physical machines without virtualization. Our current work follows a very different approach: in Koordinator the coordination is made as a service, instead of being integrated in the container management environment. In this sense, it runs outside of Kubernetes, providing transparency to the client application and improving modularity. To the best of our knownledge, there are no other works bringing together SMR and Kubernetes.

There is a recent excitement with consensus and state machine replication in the context of cryptocurrencies and blockchain [27], [28], [29]. These systems use consensus to build a replicated ledger, which is essentially a log of transactions. The original blockchain was part of the Bitcoin cryptocurrency design and provided only eventual consensus, although it tolerated Byzantine faults [27]. More recent blockchains like Hyperledger Fabric already provide strong



(a)



(b)

Fig. 7: CPU (a) and network (b) usage in experiment e2.

consensus [29]. Our approach could be used to implement a containerized blockchain with low effort.

VI. Final Remarks

This paper presented Koordinator, a first effort to incorporate coordination in Kubernetes as a service, that is, outside of container's management layer. Coordination is a necessary feature for applications that need to keep their state consistent. To evaluate the feasibility of Koordinator, some experiments were done. These experiments analyzed applications with strong consistency and eventual consistency, two semantics very employed to the internet applications.

Acknowledgments: This work was supported by the FCT via projects PTDC/EEI-SCR/1741/2014 (Abyss) and UID/CEC/50021/2013, and by the CNPq/Brasil via project 401364/2014-3.

References

[1] P. Mell and T. Grance, "The NIST definition of cloud computing," National Institute of Standards and Technology Gaithersburg, Tech. Rep., 2011.

[2] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[3] J. E. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, 2005.

[4] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," in *Proceedings of the 2nd ACM SIGOPS European Conference on Computer Systems*, 2007, pp. 275–287.

[5] D. Bernstein, "Containers and cloud: From LXC to docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[6] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the 10th European Conference on Computer Systems*, 2015, pp. 18:1–18:17.

[7] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *ACM Queue*, vol. 14, no. 1, pp. 10:70–10:93, 2016.

[8] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[9] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[10] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Annual Technical Conference*, 2014, pp. 305–320.

[11] M. Castro, B. Liskov *et al.*, "Practical Byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[12] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, "Efficient Byzantine fault tolerance," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, 2013.

[13] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with bft-smart," in *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 355–362.

[14] H. Cui, R. Gu, C. Liu, T. Chen, and J. Yang, "Paxos made transparent," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 105–120.

[15] P. Felber and P. Narasimhan, "Experiences, strategies, and challenges in building fault-tolerant CORBA systems," *Transactions on Computers*, vol. 53, no. 5, pp. 497–511, 2004.

[16] A. N. Bessani, L. C. Lung, and J. da Silva Fraga, "Extending the UMIOP specification for reliable multicast in CORBA," in *Proceedings of the nternational Symposium on Distributed Objects and Applications*, 2005, pp. 662–679.

[17] H. V. Netto, L. C. Lung, M. Correia, A. F. Luiz, and L. M. S. de Souza, "State machine replication in containers managed by Kubernetes," *Journal of Systems Architecture*, vol. 73, pp. 53–59, 2017.

[18] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "Exploiting the internet inter-ORB protocol interface to provide CORBA with fault tolerance." in *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems*, 1997, pp. 6–15.

[19] P. A. Felber, B. Garbinato, and R. Guerraoui, "The design of a CORBA group communication service," in *Proceedings of the 15th Symposium on Reliable Distributed Systems*, 1996, pp. 150–159.

[20] R. Guerraoui and A. Schiper, "The generic consensus service," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 29–41, Jan. 2001.

[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *Proceedings of the USENIX Annual Technical Conference*, 2010, p. 9.

[22] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[23] C. Dwork, N. A. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of ACM*, vol. 35, no. 2, pp. 288–322, 1988.

[24] A. Basu, B. Charron-Bost, and S. Toueg, "Simulating reliable links with unreliable links in the presence of process crashes," in *Proceedings of the 10th International Workshop on Distributed Algorithms*, 1996, pp. 105–122.

[25] E. Torresini, L. A. Pacheco, E. A. P. Alchieri, and M. F. Caetano, "Aspectos praticos da virtualizacao de replicacao de maquina de estado," in *Workshop on Cloud Networks & Cloudscape Brazil, Congresso da Sociedade Brasileira de Computacao*, 2016.

[26] Y. Amir, B. A. Coan, J. Kirsch, and J. Lane, "Prime: Byzantine replication under attack," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 4, pp. 564–577, 2011.

[27] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[28] M. E. Peck, "Blockchains: How they work and why they'll change the world," *IEEE Spectrum*, vol. 54, no. 10, pp. 26–35, 2017.

[29] J. Sousa, A. Bessani, and M. Vukolic, "A Byzantine fault-tolerant ordering service for Hyperledger Fabric," in *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2018.