Journal of Systems Architecture 000 (2016) 1–7

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

State machine replication in containers managed by Kubernetes

Hylson V. Netto[a,c,*], Lau Cheuk Lung[a], Miguel Correia[b], Aldelir Fernando Luiz[c], Luciana Moreira Sá de Souza[a]

[a] *Universidade Federal de Santa Catarina Campus Reitor João David Ferreira Lima, Florianopolis, Santa Catarina, s/n. Brazil*
[b] *INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal*
[c] *Instituto Federal Catarinense, Campus Blumenau, Blumenau, Santa Catarina, Brazil*

ARTICLE INFO

Article history:
Received 20 April 2016
Revised 15 December 2016
Accepted 23 December 2016
Available online xxx

MSC:
00-01
99-00

Keywords:
State machine replication
Shared memory
Container
Kubernetes
Docker

ABSTRACT

Computer virtualization brought fast resource provisioning to data centers and the deployment of pay-per-use cost models. The system virtualization provided by containers like Docker has improved this flexibility of resource provisioning. Applications that require more restrictive agreement and ordering guarantees can also benefit from operating inside containers. This paper proposes the integration of coordination services in a container management system called Kubernetes (k8s), seeking to restrict the containers' size and to offer automatic state replication. A protocol that uses shared memory available in Kubernetes was developed, and an evaluation was conducted to show the viability of the proposal.

1. Introduction

Virtual machines [1] enable dynamic resource provisioning in data centers. As a consequence, users can pay only for allocated resources. Services in data centers are being more consumed as more users connect to the Internet. These services are often referred to as *cloud computing*, defined by NIST [2] as "a model for enabling ubiquitous, convenient, on-demand network access to [...] computing resources [...] rapidly provisioned and released". Containers provide faster resource allocation in comparison to virtual machines [3]. Among the implementations of containers, Docker is probably the most adopted [4].

To drive the adoption of containers in clouds, companies founded the *Cloud Native Computing Foundation* (CNCF) [5]. The goals of CNCF include creating standards for container operation. Google had used containers for years [6] and launched Kubernetes (also known as *k8s*) [7] as an initial result from CNCF. Kubernetes is an open source management system for containers and counts with the knowledge of the engineers that created Borg [6], a container manager developed by Google.

Kubernetes can replicate containers to improve the availability of applications. When a container fails, Kubernetes recreates it from a predefined image. However, the state of the failed container is not restored. Applications can use external volumes to maintain their state, but it is necessary to protect the volumes against failures. Furthermore, when providing state replication, applications have to handle concurrency when accessing the volume.

Algorithms derived from Paxos [8] have been proposed to replicate services in sets of machines. Raft [9] and BFT-SMaRt [10] are used to implement State Machine Replication (SMR) [11]. Replicas communicate and follow a protocol to ensure that operations are executed on all replicas in the same order. These solutions can be applied in Kubernetes at application level, i.e., inside containers. However, each replicated container has to carry the agreement software and manage the interaction with it. To implement SMR in existing applications is not a trivial task [12].

We enable coordination in Kubernetes using *integration*, an approach that is transparent to users and shows better performance than other strategies [12,13]. Integration means building or modifying a system to complement features. In this paper we present a scheme to integrate coordination in Kubernetes. A component of Kubernetes is used as shared memory to guide the coordination.

* Corresponding author.
E-mail addresses: hylson.vescovi@posgrad.ufsc.br (H.V. Netto), lau.lung@ufsc.br (L.C. Lung), miguel.p.correia@tecnico.ulisboa.pt (M. Correia), aldelir.luiz@blumenau.ifc.edu.br (A.F. Luiz), luciana@pangeaware.com (L.M. Sá de Souza).

http://dx.doi.org/10.1016/j.sysarc.2016.12.007
1383-7621/© 2016 Elsevier B.V. All rights reserved.

We created a protocol named DORADO (or **D**ering **O**ve**R** sh**A**re**D** mem**O**ry) to do state replication in containers.

The paper is organized as follows. Section 2 presents concepts about containers and Kubernetes. Section 3 presents our coordination scheme. Section 4 evaluates the proposal, and Section 5 discusses related work. Finally, Section 6 concludes the paper.

2. Containers and Kubernetes

Containers are virtual machines (VM) instantiated from static images. Containers are normally stateless, in the sense that their state is lost when they are turned off. The images are frequently small, because only files that do not exist in its host are effectively stored in the container image. It is possible to do so with the use a layered file system, such as AuFS [14]. This way container creation becomes faster and resource provisioning happens more efficiently in comparison to traditional VMs [3]. An example of a container manager is Docker [4].

The advantage of containers over traditional virtual machines was soon perceived by Google, a company that counts with 10 years of experience with containers [15]. In high-scale environments, where many containers are used, it is essential to have a platform to manage them. Kubernetes is a container management system developed as an evolution of Borg [6]. Other companies, e.g., Red Hat, are also contributing for the development of Kubernetes, which is available in GitHub. Kubernetes is the first result of the Cloud Native Computing Foundation (CNCF), an effort to guide the adoption of containers in clouds.

Kubernetes inherits concepts from Borg [6]. For example, a web server produces logs, which can be read by a log analyzer. These applications can stay in different containers, which is useful for software updating. However, highly coupled containers should stay on the same machine to improve communication. A Borg feature called *Alloc* maintains containers on the same machine. In Kubernetes, a component named POD has the same objective as *Alloc*.

A Kubernetes cluster is composed of machines, which can be virtual or physical (Fig. 1). Each machine is a *node*. A POD is a minimal management unit and can accommodate one or more containers. 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. 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. Each node has a proxy installed. If a POD is replicated, the *proxy* distributes the load among the replicas. The *kubelet* manages PODs, containers, images and other elements in the node. The *kubelet* forwards data about the monitoring of containers to the main node, which acts when necessary.

The management components of Kubernetes are in the *main* node. The *distributed watchable storage* is implemented by the *etcd* service[1], which can notify other components when events such as data creation or update happen. Components in Kubernetes access stored data via REST API. A human operator can interact with the cluster via the *kubectl* interface command, e.g., to check the health of the cluster, or to create a POD. Other management components of Kubernetes are out of the scope of this paper.

3. Coordination in Kubernetes

Kubernetes can replicate containers, creating or destroying them, keeping a predefined number of replicas running. The load balance feature improves availability. However, some applications require stronger agreement and ordering guarantees. In collaborative applications, for example, the actions of group members have

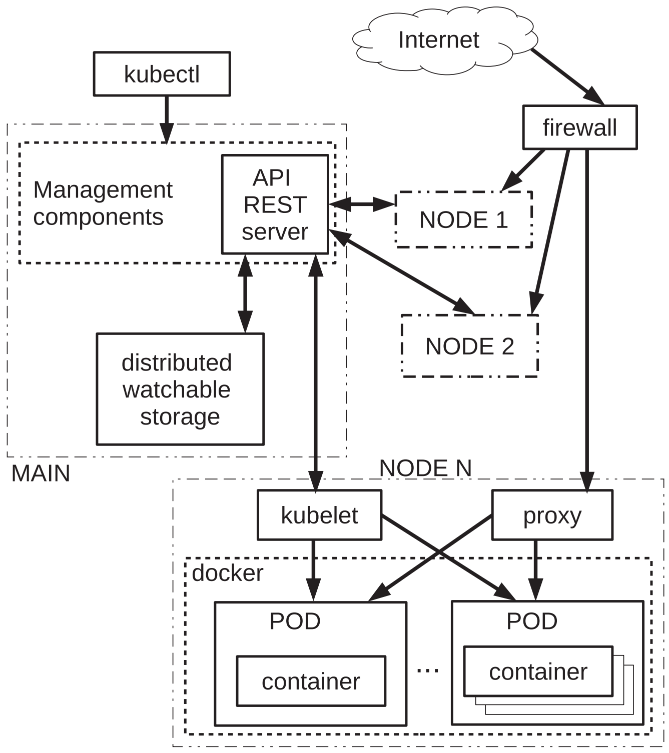


Fig. 1. Kubernetes architecture.

to be coordinated. Also, in applications that replicate data to provide fault tolerance, agreement performs a central role. Protocols such as Raft [9] and BFT-SMaRt [10] can provide coordination at application level. Nevertheless, removing the responsibility of coordination from the application can reduce the size of the container image and the complexity of the application. It can also improve the maintainability of the coordination software.

One characteristic of agreement protocols is the frequent exchange of messages between members of the group. State replication protocols implemented using shared memory (SM) can benefit from fewer message exchanged. Members of a group can only read and write values in the SM instead of multi-casting them to all members. Kubernetes uses *etcd* to maintain information about containers. This storage component can be used as a SM to implement SMR. In our proposal, we incorporate coordination in Kubernetes via integration: components can be modified or new ones can be added to the system. A similar modification was proposed when a consortium of companies created the FT-CORBA specification [13].

We propose to integrate coordination (CC) in Kubernetes with the support of an already existing component and providing a new algorithm to coordinate the requests. This architecture is shown in Fig. 2(a). In this scenario, the request is delivered by the proxy to a replicated container (3). Before executing the request, the coordination interacts with the SM (*etcd*) through a replication protocol (4). Each replicated container executes the request. Only the container that received the request replies to the client (5), via firewall (6).

An alternative solution – that we do not explore further in the paper – would be a full integration achieved by moving the coordination (CC) to other component in Kubernetes, or creating a new one (Fig. 2(b)). The client sends the request (1), which is delivered to CC (2). A consensus protocol (DORADO, presented in this paper, or another consensus protocol) orders the request (3), being executed in all containers (4). The CC that received the request sends the answer (5), which is delivered to the client by the firewall (6).

[1] https://coreos.com/etcd

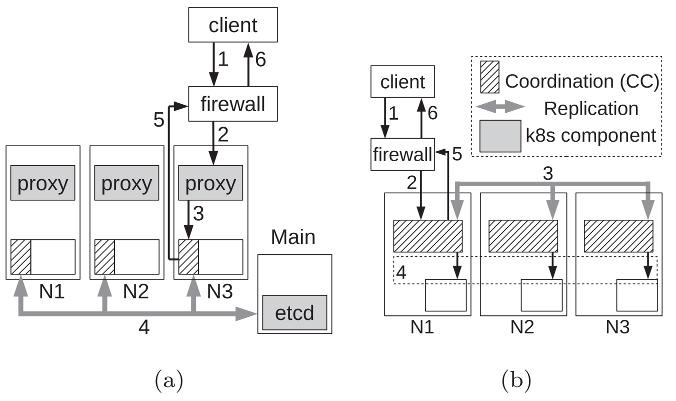


Fig. 2. Kubernetes with integrated coordination inside: (a) containers, (b) a new component. Our approach is (a); the other one is not explored in this paper.

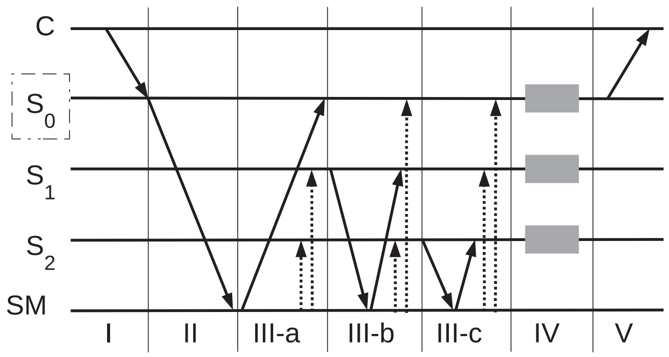


Fig. 3. DORADO protocol, request to leader.

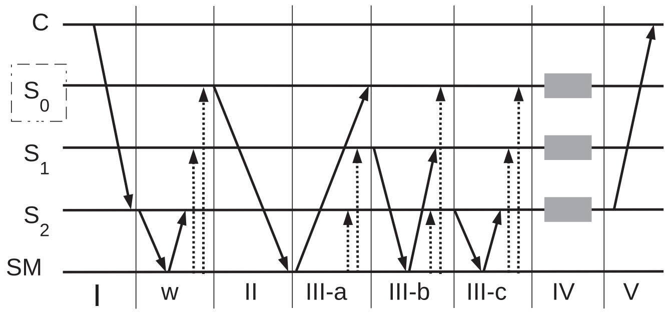


Fig. 4. DORADO protocol, request to a non-leader.

3.1. System model

There is a set $\mathcal{S}$ of replicated containers[2], out of which a maximum of $f < |\mathcal{S}|/2$ can fail by crash. Containers do not recover from failures, but new containers can be launched to replace failed ones, reconfiguring the system [11]. An arbitrary (but finite) number of clients can access the system. Communication among containers happens via shared memory (SM), implemented by the *etcd* service. The SM is a key-value repository accessible via read/write operations. It can notify subscribing processes about events on data (e.g., updates). Another feature of the SM is the association of an unique identifier to each data item it stores. The SM can be distributed with Raft [9], improving its tolerance to crash faults, at cost of higher load on the network and delay.

Interaction between clients and containers is assumed to be partially synchronous [16], while communication between containers and the components of Kubernetes (SM and the API server) is asynchronous. Containers have their state replicated by executing all requests in the same order [11]. A coordination protocol establishes the requests order (Section 3.2).

3.2. State replication protocol

In order to maintain states replicated in containers, at least two properties must be satisfied:

- safety: every non-faulty container executes requests it receives in the same relative order.
- liveness: every request is eventually executed by non-faulty containers.

A requirement to be considered is that any replica must be able to receive requests from the client. This is important because Kubernetes distributes load at node level, via the *proxy* component.

We designed a protocol named DORADO (orDering OveR shAreD memOry) to order requests in Kubernetes. Fig. 3 presents an execution of DORADO in a scenario with three replicas ($S_{0..2}$). One crash fault is tolerated ($f = 1$). Replica S_0 is the leader (box with dashed line) and receives the request sent by the client. In the request phase (I), the client sends the request to some replica. If this replica is the leader (Fig. 3), it proposes an order to the request (II), saving it in the shared memory. The leader replica is the only one which has the authority to define order for requests. All replicas will execute requests following this defined order. The SM confirms the saving and notifies other replicas (III-a). When replicas receive the notification of a request, they accept the request and save their accepts in SM (III-b and III-c). After a replica collects accepts from a majority of replicas, it learns the request (IV) and executes it. The replica that received the request informs the client of the result (V).

Note that at the end of phase III-b, replicas S_0 and S_1 have collected accepts from a majority of replicas (including their own accepts). That way phase III-c could be ignored, which means that a crash of replica S_2 could be tolerated. A fault of replica S_1 could be tolerated if phase III-b did not happen, since phase III-c would run. If the client does not receive the answer after some time (defined by a timeout), it sends the request again to another replica.

If the client sent the request to a non-leader replica S_2 (I), one extra phase is required (Fig. 4). Next, phase *w*, S_2 creates an accept with order zero, and saves this accept in SM. Replicas will be notified, but only the leader will act. The leader orders the request and, from that point on (phase II), the protocol flows as depicted in Fig. 3. Finally, in phase V replica S_2 sends the reply to the client.

Replicas have to wait for a majority of accepts before replying to the client. This happens because during view changes a new leader has to define the order of requests considering the last request already ordered and executed. As in a membership service it is necessary that every two consecutive views intersect [17] to satisfy the safety property (Section 3.1), a replica can only reply to the client when it is aware that a majority of replicas already knows that answer. The consistency of the system is satisfied because the intersection of any majority quorum with the overall group contains at least one element of the minority group. If a minority number of replicas fail, consistency and liveness are ensured.

3.2.1. Leader election

Protocols that use sequencers have to use timers to check the leader's progress [18]. In DORADO, timers are activated when replicas realize the existence of a request *P*. This can happen when a replica receives *P* from the client or when a replica receives notifications about *P* from the SM. When a timer expires and *P* remains unordered, the replica will save a leader election solicitation (LESO)

[2] In this paper we use the term *container* instead of POD.

in SM. The SM notifies the replicas, which will save LESOs in the SM and create new notifications.

When a replica collects LESOs from a majority of replicas, the election can be decided. The new leader will be the first replica that saved a LESO in SM. To avoid possible network message inversions and to deal with concurrency, the replica reads from SM the complete set of LESOs. This way, it is possible to define precisely the first replica that voted for election.

3.2.2. *Replication over* etcd

In this section we clarify some aspects about the execution of DORADO over *etcd*. The *etcd* service includes the Raft algorithm, which can be enabled to replicate the service and ensure the consistency of the data it stores. Kubernetes accesses *etcd* through the API Server component, which deals with concurrent requests that come from other components (e.g., kubelets). DORADO accesses *etcd* to persist requests coming from clients, and to broadcast messages to replicas. When executing SMR, the concurrency rule is that all replicated containers have to execute all requests in the same order. In this context, the ordering provided by *etcd* is not enough to define the order of requests. Consider two facts:

- *Fact 1*: The notification mechanism of *etcd* does not deliver messages using total order.
- *Fact 2*: The UID generated (by *etcd*) for values under a key is monotonic but is not ensured to be sequential. Data created in other keys can create jumps in the UID perceived by a client listening for updates in a key.

When clients send requests concurrently, the UIDs of data saved in *etcd* (propose, accept) could be used to decide the order of requests. However, because of Fact 1, replicas might be notified with gaps between the received UIDs. It is not possible to wait for notifications inside the gaps because of the Fact 2 (how many notifications should be expected?). We solved this issue in leader elections by reading all storage before the definition of the new leader (Section 3.2.1), but this strategy is only suitable because election is not a frequent operation. It is not reasonable to apply this solution to order requests because of the high volume of data (proposes, accepts) to be reloaded.

DORADO uses the UID to decide election (Section 3.2.1). The UID consistency is maintained by Raft when the storage is replicated. Consequently, only during elections Raft is essential for ensuring the correctness of elections in DORADO. In normal operation DORADO has its own leader which defines the order of requests.

4. Evaluation

In this section we evaluate DORADO. Preliminary tests indicated a high activity in the hard disk of the main node. This activity came from the usage of *etcd* as SM. In order to reduce latency, a virtual disk was mounted in RAM and *etcd* was configured to store data on the virtual disk. Although persistence could be compromised with a virtual disk, similar benefits (e.g., lower latency) might be obtained with SSD disks. A *first question* to answer is if the usage of SM on a virtual disk (RAM) will provide lower latency than the usage of the SM on a hard disk.

Resource allocation is faster using containers than using virtual machines [3]. Also, the usage of shared memory in DORADO simplifies the communication, which can benefit network load when the number of replicas increases. That way, we can conjecture the throughput of the system will not fall down abruptly when scaling up the number of replicas. As a *second question* we evaluate if the throughput decreases gradually as more replicas enter in the system.

4.1. *Environment and implementation*

The experiments were executed in a cluster of four computers with Kubernetes 1.1.7. A fifth computer acted as client. All computers had CPUs Intel i7 3.5 GHz, QuadCore, cache L3 8MB, 12GB RAM, and 1TB HD 7200 RPM. An Ethernet network 10/100 Mbits isolated from external traffic was used to connect the computers. Ubuntu Server 14.04.3 64 bits was used, with kernel 3.19.0-42.

The DORADO protocol was implemented in Go language, version 1.4.2. The evaluated application maintains a shared counter. Three commands were used (in a round-robin fashion): *get* and *inc* return and increase the value, respectively, and *dou* divides the counter by 2 if the value of the counter is greater than 30. Our application is called *replicatedcalc*. The container image is available in Docker Hub.[3] The source code is in GitHub.[4]

Limitations: DORADO uses a garbage collector in the SM to avoid the unlimited growth of the data and a checkpointing mechanism to limit the amount of messages required to restore the state. However, the current prototype does not contain these two mechanisms. Therefore the experiments measure the performance between them and garbage collection actions, with would normally be executed periodically (e.g., every several minutes).

Raft was not active in the experiments so *etcd* was not distributed (it was being executed only in the main node of Kubernetes). However, if running, Raft can batch messages to improve performance and reduce network communication. Batching means to gather multiple requests in one message. This feature is available in the implementation of Raft used in *etcd*.[5]

4.2. *Experiments*

To investigate the first question, the main node of Kubernetes was provided with a virtual disk of 4GB in RAM (*tmpfs* file system). We executed the experiment using *etcd* with hard disk and virtual disk. The second question motivates variations in the number of replicas (1 to 15), with $n = 2f + 1$. The number of clients was varied from 1 to 64, doubling in each variation. We measured the latency perceived by each client in milliseconds and the duration of each experiment in seconds. Multiple threads simulated multiple clients accessing the system. The requests and replies had approximately 100 bytes. Each client made 50 requests to the system. Clients only send a new request after receiving the reply to the previous one.

Latency increases as more replicas enter the system (Fig. 5). Solid bars represent execution with RAM disk and transparent bars represent execution with hard disk. As expected, latency is lower when using RAM disk. However, as more clients access simultaneously the system, the difference of latency decreases. We present an explanation for this behavior. Hard drives have rotating disks, so the heads have to move to handle random access, causing high latencies [19]. Although we use *etcd* mostly for writing, during the experiments we noted that *etcd* did not use cache, flushing all write operations.[6] To ensure persistence, extra time is required, even for 2 or 32 simultaneous requests. This overhead is noticeable when few clients access the system. For many simultaneous accesses (more than 16 clients in our experiments), the additional latency overlaps the time to process many requests, not interfering with the total latency, for both media used in the SM.

Following latency trends, the media used in SM has lower influence in throughput as more clients access the system (Fig. 6). Throughput decreases as more replicas enter the system.

[3] https://hub.docker.com/r/hvescovi/replicatedcalc/
[4] https://github.com/hvescovi/learnGo/
[5] Cf. http://github.com/coreos/etcd/tree/master/raft
[6] We checked the *dirty* information at /proc/meminfo.

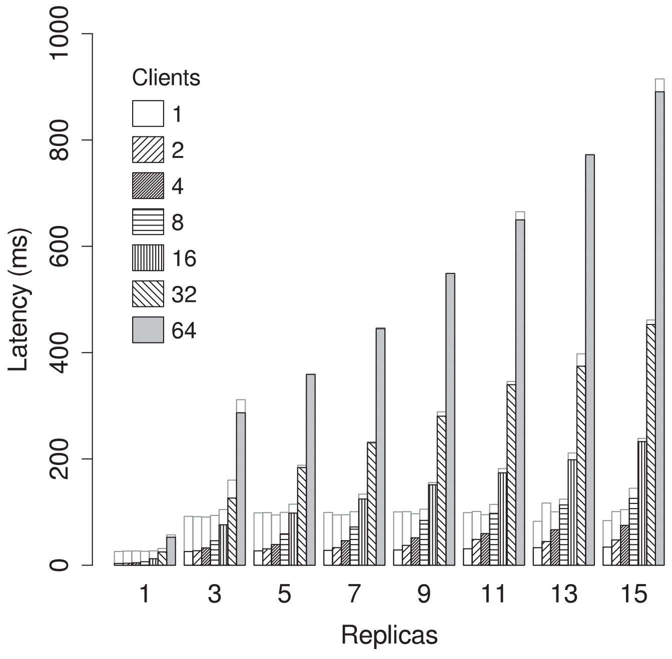


Fig. 5. Latency observed by clients.

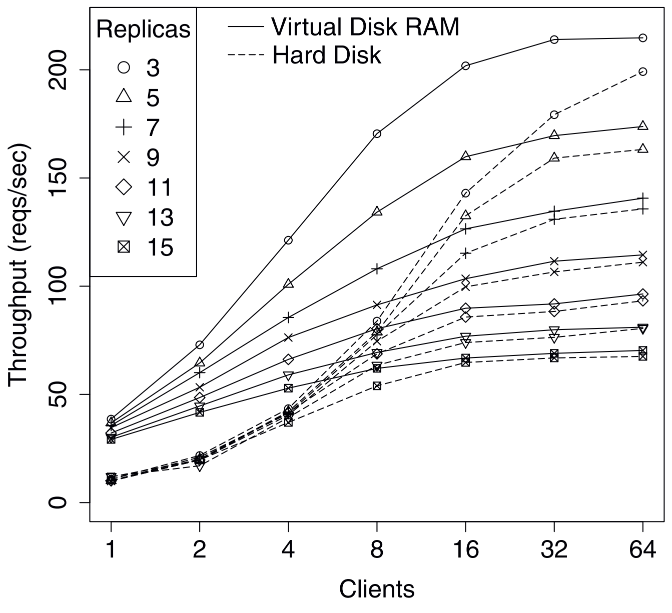


Fig. 6. Throughput of the system.

5. Related work

There is a long literature in fault tolerance for crash failures. However, to the best of our knowledge, this work is the first to provide crash fault tolerance and coordination taking into account the lightweight nature of containers as virtual machines. We developed a light protocol which leverages an already existing component. That way, the code inside containers (and the size of the container image) can be reduced. We discuss some recent work on fault-tolerant replication and describe the aspects closer to our proposal [9,12,20–22].

Systems can be improved with reliable components. For example, *wormholes* can be used as trusted components to provide reliable message broadcast [20]. In BAMCast [22], a reliable channel among replicas is created using distributed shared registers. This channel has bounded delay and is used to execute critical parts of the BAMCast consensus protocol. Both works reduced the number of replicas required to provide Byzantine fault tolerance. In our proposal, the usage of shared memory allowed the development of a simpler protocol. The SM is an already existing component in Kubernetes that uses Raft to tolerate crash faults.

The development of a protocol in a layered fashion creates a decoupling between components, allowing the technological evolution of these same components. For example, virtual machines (VM) can provide a SM abstraction by sharing folders, although this reduces fault tolerance [22,23]. In that case, if some VM are inside the same host, less communication among the nodes network will be required to keep the SM consistent. Notice that typically this host (main node) will be different from the hosts where DORADO is executed (nodes).

Gaios [21] is an enhanced version of Paxos. The main improvement consists in organizing disk access efficiently. Cache is used for writing in-memory, data being written to the disk in a disk-efficient order at checkpoint times. However, the state can be damaged if many faults occur simultaneously. In DORADO all requests are persisted in SM before execution, ensuring durability and consistency despite major failures. Furthermore, in our experiments, the disk had impact on performance only in low contention scenarios.

Raft [9] is a consensus protocol designed to be easy to understand and implement. In Raft the registration of new requests (append of entries in log) follows only from the leader to other replicas. In DORADO, not only the leader can receive requests, but any replica. Raft uses randomized timeouts in leader election and the heartbeat mechanism simplifies conflicts resolution. Leader election in DORADO has deterministic criteria, and heartbeats are not required.

CRANE [12] makes Paxos transparent for applications. It intercepts requests to execute them on the same order in all replicas. DORADO proposes a transparent coordination using integration, instead of interception. The integration approach is used in this paper because it presents a better performance than other approaches (interception and service) [13].

6. Conclusions

This paper presents the DORADO protocol, a first step for integrating coordination services in Kubernetes, a container management system. DORADO's design is simpler than others as it uses share memory to mediate communication and persist data. Requests can be sent to any replicated container, matching the cloud nature in regard to load balancing.

Future steps in this work include moving coordination to a component in Kubernetes, which will actually reduce the size of containers. This could be done moving the coordination to the *proxy*, or using a leaderless protocol as EPaxos [24] to avoid the necessity of choosing a proxy-leader. Another improvement would be to take the leader election out of the SM, so that it runs separately from *etcd* and Raft is no longer needed. Kubernetes itself can be used for doing leader election.[7]

The extension of DORADO to the Byzantine domain should be a challenge, because there are issues to be solved about isolation. Additional security layers are required to protect containers against malicious actors [15]. However, the execution of containers inside VM could result in excessive overheads.

DORADO could be implemented in systems like Apache Mesos[8] and Docker Swarm[9], which use ZooKeeper and Consul,

[7] http://blog.kubernetes.io/2016/01/simple-leader-election-with-Kubernetes.html
[8] http://mesos.apache.org
[9] http://www.docker.com/products/docker-swarm

respectively, to maintain the state of the cluster. ZooKeeper and Consul have similar features to *etcd*. ZooKeeper can be used to obtain sequential numbers with the sequential flag that provides unique znode names [25]. This feature might be leveraged to develop a new protocol to define the ordering of requests based on destination agreement [18].

Acknowledgments

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 and CNPq 455303/2014-2. This work was partially supported by FAPESC/IFC through project nº 00001905/2015. Miguel Correia is bolsista CAPES/Brasil (project LEAD CLOUDS).

Sadly during the final revision of this paper the second author, our advisor, colleague, and friend Lau Cheuk Lung was deceased due to sudden health complications. We take this opportunity to pay a small homage to a great researcher and professor who influenced many students and colleagues during his career.

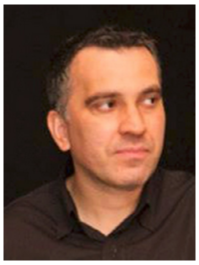
References

[1] G.J. Popek, R.P. Goldberg, Formal requirements for virtualizable third generation architectures, Commun. ACM 17 (7) (1974) 412–421.
[2] P. Mell, T. Grance, The NIST definition of cloud computing, Technical Report, National Institute of Standards & Technology, 2011. SP 800-145.
[3] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An updated performance comparison of virtual machines and Linux containers, in: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2015, pp. 171–172.
[4] R. Peinl, F. Holzschuher, F. Pfitzer, Docker cluster management for the cloud – survey results and own solution, J. Grid Comput. (2016) 1–18.
[5] A. Sill, Emerging standards and organizational patterns in cloud computing, IEEE Cloud Comput. 2 (4) (2015) 72–76.
[6] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes, Large-scale cluster management at Google with Borg, in: Proceedings of the 10th European Conference on Computer Systems, 2015.
[7] D. Bernstein, Containers and cloud: from LXC to Docker to Kubernetes, IEEE Cloud Comput. 1 (3) (2014) 81–84.
[8] L. Lamport, The part-time parliament, ACM Trans. Comput. Syst. 16 (2) (1998) 133–169.
[9] D. Ongaro, J. Ousterhout, In search of an understandable consensus algorithm, in: Proceedings of the USENIX Annual Technical Conference, 2014, pp. 305–320.
[10] A. Bessani, J. Sousa, E.E.P. Alchieri, State machine replication for the masses with BFT-SMART, in: Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2014, pp. 355–362.
[11] F.B. Schneider, Implementing fault-tolerant services using the state machine approach: a tutorial, ACM Comput. Surv. 22 (4) (1990) 299–319.
[12] H. Cui, R. Gu, C. Liu, T. Chen, J. Yang, Paxos made transparent, in: Proceedings of the ACM Symposium on Operating Systems Principles, 2015, pp. 105–120.
[13] A.N. Bessani, J. da Silva Fraga, L.C. Lung, E.A.P. Alchieri, Active replication in CORBA: standards, protocols, and implementation framework, in: On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE, 2004, pp. 1395–1412.
[14] D. Merkel, Docker: lightweight Linux containers for consistent development and deployment, Linux J. 2014 (239) (2014).
[15] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, J. Wilkes, Borg, Omega, and Kubernetes, Commun. ACM 59 (5) (2016) 50–57.
[16] C. Dwork, N. Lynch, L. Stockmeyer, Consensus in the presence of partial synchrony, J. ACM 35 (2) (1988) 288–323.
[17] G.V. Chockler, I. Keidar, R. Vitenberg, Group communication specifications: a comprehensive study, ACM Comput. Surv. 33 (4) (2001) 427–469.
[18] X. Défago, A. Schiper, P. Urbán, Total order broadcast and multicast algorithms: taxonomy and survey, ACM Comput. Surv. 36 (4) (2004) 372–421.
[19] F. Chen, D.A. Koufaty, X. Zhang, Understanding intrinsic characteristics and system implications of flash memory based solid state drives, ACM SIGMETRICS Perform. Eval. Rev. 37 (2009) 181–192.
[20] G.S. Veronese, M. Correia, A.N. Bessani, L.C. Lung, P. Verissimo, Efficient Byzantine fault tolerance, IEEE Trans. Comput. 62 (1) (2013) 16–30.
[21] W.J. Bolosky, D. Bradshaw, R.B. Haagens, N.P. Kusters, P. Li, Paxos replicated state machines as the basis of a high-performance data store., in: Proceedings of the Symposium on Networked Systems Design and Implementation, 2011, pp. 141–154.
[22] M.R.X. Silva, L.C. Lung, L. Q. Magnabosco, L. O. Rech, BAMcast: byzantine fault--tolerant consensus service for atomic multicast in large-scale networks, in: Proceedings of the IEEE Symposium on Computers and Communications, 2013, pp. 324–329.
[23] F. Dettoni, L.C. Lung, M. Correia, A.F. Luiz, Byzantine fault-tolerant state machine replication with twin virtual machines, in: Proceedings of the IEEE Symposium on Computers and Communications, 2013, pp. 398–403.
[24] I. Moraru, D.G. Andersen, M. Kaminsky, There is more consensus in egalitarian parliaments, in: Proceedings of the ACM Symposium on Operating Systems Principles, 2013, pp. 358–372.
[25] P. Hunt, M. Konar, F.P. Junqueira, B. Reed, Zookeeper: wait-free coordination for internet-scale systems., in: Proceedings of the USENIX Annual Technical Conference, 2010.

Hylson Vescovi Netto is graduated in Computer Engineering (2001), and received the MsC Degree in Electrical Engineering (2003). He is currently pursuing his PhD studies at UFSC. His main research is about Distributed Systems and Fault Tolerance. He is professor at Instituto Federal Catarinense since 2005.

Lau Cheuk Lung received the PhD degree in electrical engineering from the Federal University of Santa Catarina (UFSC), Brazil, in 2001. He is an associate professor in the Informatics and Statistic Department (INE), UFSC. He is currently doing research in fault tolerance, security in distributed systems, and middleware. From 2003 to 2007, he was an associate professor in the Department of Computer Science, Pontifical Catholic University of Paraná, Brazil. From 1997 to 1998, he was an associate research fellow at the University of Texas at Austin, working in the Nile Project. From 2001 to 2002, he was a postdoctoral research associate in the Department of Informatics, University of Lisbon, Portugal.

Miguel Correia is an Associate Professor at Instituto Superior Técnico (IST) of the Universidade de Lisboa (ULisboa), in Lisboa, Portugal. He is a researcher at INESC-ID in the Distributed Systems Group (GSD). He is currently the coordinator of the Master Degree (MSc) in Information Systems and Computer Engineering. He has a PhD in Computer Science from the University of Lisboa Faculty of Sciences. He has been involved in several international and national research projects related to intrusion tolerance and security, including the SafeCloud, PCAS, TCLOUDS, ReSIST, CRUTIAL, and MAFTIA European projects. He has more than 100 publications and is Senior Member of the IEEE. His main research interests are: security, intrusion tolerance, distributed systems, distributed algorithms, computer networks, cloud computing, and critical infrastructure protection.


Aldelir Fernando Luiz is graduated in Computer Science (2001), with specialization in Computer Networks and Distributed Systems at UFSC. He has the Master Degree in Informatics (2009) and the PhD Degree in Engineering of Automation and Systems (2015). He worked as consultor of database and information systems at TOTVS S/A during 10 years and has an Oracle OCP certification. Now he is professor at Instituto Federal Catarinense. His main research intererests are in Distributed Systems, Fault Tolerance, Distributed Algorithms, Data bases and Information Systems.

Luciana Moreira Sá de Souza is graduated in Engineering of Automation and Control (2002) and has the Master Degree in Electrical Engineering (2005), both at UFSC. She obtained the PhD Degree in Fakultät für Informatik pela Universität Karlsruhe(2009). Her areas of interest are Distributed Systems, Fault Tolerance and Wireless network sensors. She is now in a Post-Doc at UFSC.