Container Allocation and Deallocation Traceability using Docker Swarm with Consortium Hyperledger Blockchain

Marco A. Marques[1][a], Charles C. Miers[1][b] and Marcos A. Simplicio Jr.[2][c]
[1]*Graduate Program in Applied Computing (PPGCAP), Santa Catarina State University (UDESC), Brazil*
[2]*Laboratory of Computer Networks and Architecture (LARC), University of São Paulo (USP), Brazil*

Keywords: Docker, Blockchain, Monitoring, Container, Traceability.

Abstract: Container-based virtualization enables the dynamic allocation of computational resources, thus addressing needs like scalability and fault tolerance. However, this added flexibility brought by containerization comes with a drawback: it makes system monitoring more challenging due to the large flow of calls and (de)allocations. In this article, we discuss how recording these operations in a blockchain-based data structure can facilitate auditing of employed resources, as well as analyses involving the chronology of performed operations. In addition, the use of a blockchain distributes the credibility of record integrity among providers, end-users, and developers of the container-based solution.

1 INTRODUCTION

A computing architecture based on microservices facilitates the construction of high performance and scalable applications, since they can be fragmented into independent parts for development, versioning, and provisioning (Jamshidi et al., 2018). Containers are commonly considered the standard of microservices in the cloud, in particular, due to their speed, ease of allocation, scalable management, and resilience (Newman, 2015). Such benefits led to the development of many container orchestration platforms. Designed to manage the deployment of containerized applications in large-scale agglomerates, such platforms can execute hundreds of thousands of jobs on different machines (Rodriguez and Buyya, 2019). As such, they became a core technology for enabling the on-demand offering of computing infrastructure, platforms, and applications by modern cloud providers.

In this context, providing tools for monitoring and auditing the execution of the virtual environment is an important requirement. After all, such tools allow all actors involved in the cloud's operation to monitor and optimize the execution of applications, track failures, configure billing modules, manage changes, among other activities (Jiménez et al., 2015). In general, cloud computing providers supply a monitoring system for all participants. Nevertheless, each actor can implement an independent monitoring service if desired (Dawadi et al., 2017). This possibility brings more autonomy to the actors, who can then customize the monitoring tools to their needs and gain further insight on the usage of resources by their applications. However, multiple independent monitoring systems can result in disagreements among actors regarding the information collected by their tools, resulting in different views for the same scenario and difficulties in later attempts of mapping events from one log into another. Centralized monitoring solutions, on the other hand, limit customization, besides depending directly on the trust placed on the system manager for protecting the integrity of the collected data.

To address this issue, this work proposes and implements a solution called event2ledger. This tool is executed by each actor of the system for (1) collecting container life cycle events using the Docker API, (2) generating and signing the corresponding transactions, and (3) sending them to a Hyperledger Fabric consortium blockchain. The blockchain is maintained by the system actors, so any of them can validate that transactions are issued by authorized parties, that the application's chaincode functions are satisfied, and storing events after reaching a consensus on their order. A proof-of-concept implementation is described for enabling the monitoring of containers'

[a] https://orcid.org/0000-0001-5800-8927
[b] https://orcid.org/0000-0002-1976-0478
[c] https://orcid.org/0000-0001-5227-7165

Marques, M., Miers, C. and Simplicio Jr., M.
Container Allocation and Deallocation Traceability using Docker Swarm with Consortium Hyperledger Blockchain.
DOI: 10.5220/0010493302880295
In *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*, pages 288-295
ISBN: 978-989-758-510-4

execution time via their life cycle events. We emphasize, though, that the same tool can be used for collecting other events available in the Docker environment.

The main contributions of this paper are: *(i)* the design of a solution that collects container events in a non-intrusive manner, sending them to Hyperledger Fabric blockchain through signed transactions; *(ii)* the implementation of a Hyperledger Fabric blockchain to verify authorizations, reach a consensus and store the collected transactions in a distributed ledger; and *(iii)* the construction of an environment that facilitates auditing and tracking anomalous (possibly malicious) behavior or disagreements about container events data collected. We emphasize, however, that the proposed solution is not intended to prevent the action of malicious agents (e.g., who omit or provide bogus information to the blockchain). Instead, the goal of using a blockchain in this scenario is to create a verifiable append-only log (Laurie, 2014), thus facilitating the tracking of conflicts and discrepancies among data collected by the event2ledger instances in the environment.

The rest of this work is organized as follows. Section 2 delimits the problem to be addressed, as well as the functional and non-functional requirements assumed for the scenario. Section 3 details the proposed solution, showing how it can address the problem of traceability in containerized environments. Section 4 presents the related work. Section 5 describes a proof-of-concept implementation of the solution. Section 6 presents the test performed and the results obtained, which are then discussed in Section 6.1. Section 7 concludes the discussion and suggests ideas for future works.

2 PROBLEM DEFINITION AND REQUIREMENTS

In cloud computing systems, monitoring tools collect data about the execution of the virtual environment and then provide a set of relevant variables for analysis by administrators and/or automated mechanisms. As discussed in Section 1, one issue in this scenario is that the deployment of multiple monitoring tools by different actors may result in conflicts between the data collected, whereas centralized monitoring hinders integrity checking in case of failures and limits the customization capabilities. Addressing these conflicts efficiently and effectively is the main goal of this work.

In this article, we are particularly interested in a container-based virtualization environment, which provides a higher degree of dynamism than regular virtual machines. To build a testbed where such conflicts among monitoring tools are observable and a solution can be deployed, one needs the following:

- A virtualization environment with a suitable containerization platform (e.g., Docker) and corresponding orchestration tool (e.g., Docker Swarm).
- A multitude of nodes, each of which playing the role of either provider, developer, or user, thus ensuring decentralization of the monitoring tasks,
- Access to a repository for data storage, in which the life cycle events collected by the solution will be stored.
- Access to the orchestration API, with permission to execute, suspend and terminate containers, to generate the life cycle events, as well as collect data on generated events.

In such an environment, developers deploy their applications in the form of multiple containers that are instantiated on-demand, according to the user's needs. Monitoring the container's execution is, thus, important to all involved actors: providers, developers, and users. Hence, any solution for this purpose must be reliable and auditable, delivering optimization and pricing capabilities in compliance with the established service level agreement (SLA). To accomplish this goal, we define the following functional (FR) and non-functional (NFR) requirements.

- FR1: Collect all creation, suspension, return, and completion events related to containers in the virtualization environment.
- FR2: Format and send the events according to the storage repository;
- FR3: Promote consensus among the actors involved in the process; and
- FR4: Store the validated records in a distributed and tamper-protected repository accessible to all actors involved in the process, thus allowing data to be audited.
- NFR1: Adjust the volume available for storage of collected events according to the size of the environment and to the number of containers running;
- NFR2: Collect events of all containers in the environment; and
- NFR3: Limit the overhead introduced by the solution, so the environment's performance is not noticeably affected.

3 PROPOSED SOLUTION

The details of container management may be quite complex and depend on the specific orchestration technology employed. Usually, the container life cycle can be summarized as shown in Fig. 1.

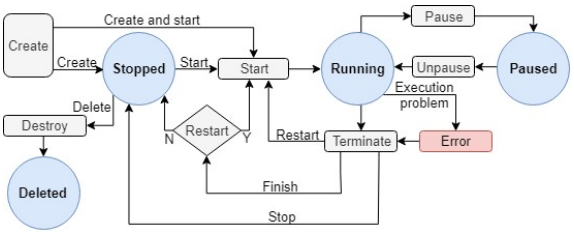

Figure 1: Docker container life cycle.

Typically, the Docker container life cycle comprises four main states: stopped, running, paused, and deleted. The transition between these states can be triggered via a command-line interface or by an orchestrator. The Docker daemon is then responsible for performing the action. Each object managed by Docker (*e.g.*, containers, volumes, networks, and images) has a set of events that can be collected and stored. As the proposed solution aims to monitor the container execution time, it focuses on events related to state changes, namely: create, connect, start, kill, die and destroy. However, it is possible to customize the solution to collect other types of events, if needed. Figure 2 provides a general overview of the proposed solution, showing how it interacts with Docker and the Hyperledger Fabric blockchain.

In Figure 2, the Provider hosts the container virtualization environment. In this environment, developers implement and offer their applications to Users, who execute them on demand, generating events (1). Every time a container has its life cycle state changed, event2ledger collects the corresponding event via Docker API (2). Then, a digitally signed transaction containing relevant information about the event is generated and sent to the blockchain while the action is performed (3). The Hyperledger Fabric blockchain first verifies if the transaction is correctly signed by the collector. Then, the applicable chaincode function is invoked and executed by the endorsing nodes. After execution, the transaction is signed and sent back to the client. When the client has enough signatures, the transaction is added to the blockchain via a Raft consensus algorithm, a Crash Fault Tolerant (CFT) implementation that supports failure in up to one-third of its nodes. Transactions are ordered and delivered to all actors (4), each of which validates the transaction and stores the results in its own ledger (5). All transactions submitted to

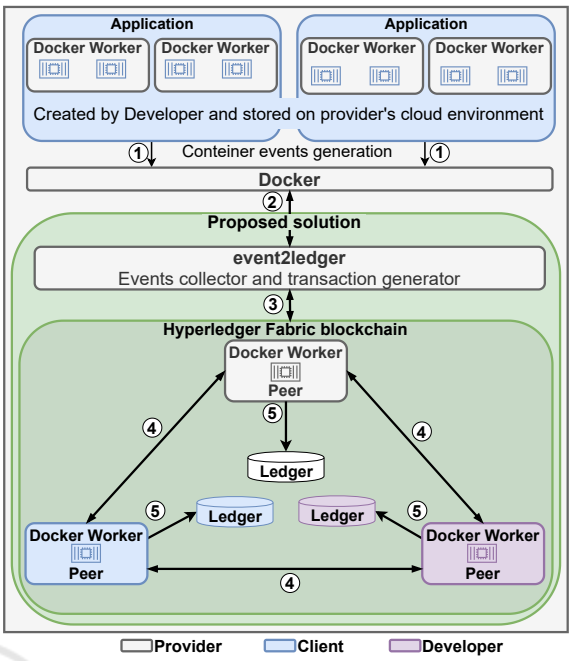

Figure 2: Proposed solution.

the blockchain are stored even if the validation phase fails; in this latter case, the transaction is labeled as failed and does not update the ledger's global state.

4 RELATED WORKS

To identify related works, we used the expression [(docker OR container) AND monitoring] in the following scientific search engines: Google Scholar, ACM Digital Library, IEEE Xplore, and Springer Link. Table 1 summarizes the inclusion and exclusion criteria adopted for the articles found in this manner.

Table 1: Inclusion and exclusion criteria.

Inclusion criteria	Exclusion criteria
Articles written in English	"Gray" literature
Articles, abstracts, book chapters, and technical reports	Publication date prior to 2012
Container monitoring in Docker environment	

When works that are extensions of another were identified, only the most recent version was considered. We note that the "gray" literature exclusion criteria refer to publications (e.g., blogs and magazines) that are not peer-reviewed and, hence, usually lack scientific rigor. Also, as a second exclusion criterion, we used the date of publication of 2012 because the Docker technology was first released into the public domain in 2013 (Hykes, 2013), and since then became one of the most successful container architectures available.

After conducting the searches and applying the fil-

Table 2: Related papers *vs.* Functional requirements.

	(Jo et al., 2018)	(Dawadi et al., 2017)	(Ciuffoletti, 2015)	(Oliveira et al., 2017)	(Pourmajidi and Miranskyy, 2018)	(Collectd, 2020)	(cAdvisor, 2020)
FR1	Yes	Yes	No	No	No	Yes	Yes
FR2	Yes	Yes	Yes	Yes	Yes	Yes	Yes
FR3	No	No	No	No	Yes	No	No
FR4	No	No	No	Yes	Yes	Yes	Yes

ters defined as inclusion and exclusion criteria, 20001 results were obtained. To identify the relevance to the proposed theme, the resulting works were analyzed, initially, by title and keywords. Those considered relevant were then selected and their abstract was analyzed. After this stage, 10 papers were selected for reading, in which 5 papers were related to the subject:

- *Light-Weight Service Lifecycle Management For Edge Devices In I-IoT Domain* (Jo et al., 2018): a solution for managing the lifecycle of containers using the Docker framework.
- *CoMMoN: The Real-Time Container and Migration Monitoring as a Service in the Cloud* (Dawadi et al., 2017): a tool for collecting and storing metrics and events, in addition to migrating Docker containers.
- *Automated deployment of a microservice-based monitoring infrastructure* (Ciuffoletti, 2015): a microservice monitoring structure "*as a service*".
- *A Cloud-native Monitoring and Analytics Framework* (Oliveira et al., 2017): a structure for collecting and storing container metrics in a distributed repository.
- *Logchain: Blockchain-assisted Log Storage* (Pourmajidi and Miranskyy, 2018): uses a blockchain-based logging system, aiming to ensure data immutability.

Table 2 presents a comparison between the functional requirements of the proposed solution and the related papers identified. FR1 is attended by (Jo et al., 2018) and (Dawadi et al., 2017), and by the Collectd and cAdvisor tools, enabling the container life cycle events to be collected in a non-intrusive way. FR2, in turn, is fulfilled by all identified papers. FR3 requires verification if there is a consensus among the actors involved in the process or if conflicts exist. The solution proposed in (Pourmajidi and Miranskyy, 2018) is the only one, among identified proposals, that tackles this requirement. Specifically, that work assumes that data blocks are inserted into a Blockchain using a consensus mechanism based on Proof of Work (PoW), so different participants can check data consistency before registration takes place. Finally, FR4 specifies that the collected data must be stored in a distributed repository. The works proposed by (Jo et al., 2018; **?**) rely on centralized storage of collected data and, thus, do not meet this requirement. Meanwhile, the scope of the proposal submitted by (Ciuffoletti, 2015) does not cover the storage of collected data. In contrast, the work presented in (Oliveira et al., 2017) uses a distributed database, whereas (Pourmajidi and Miranskyy, 2018) employs a blockchain-based model, meeting the requirement. Finally, CollectD and cAdvisor natively support data exportation to various repository models, including distributed ones, besides allowing the development of new integration plugins.

In summary, the analysis of related works indicates that none of them fully satisfies our FRs. Actually, the solution presented in (Pourmajidi and Miranskyy, 2018) is the one that comes closest, but it was designed for the monitoring of computational clouds in general, so it does not take care of the fine granularity and high dynamism involved in the monitoring of container events. The open-source tools cAdvisor and Collectd, on the other hand, are particularly interesting for their capability of storing data in distributed repositories, but none of them give on-the-fly visibility to conflicts in the collected events (e.g., via a consensus mechanism).

5 IMPLEMENTATION

The proposed solution has two components: (1) event2ledger, responsible for collecting events and generating transactions, and (2) a blockchain Hyperledger Fabric, whose nodes belong to the actors involved in the microservices ecosystem. Both components are implemented in a Docker container environment, over a GNU/Linux Ubuntu desktop 20.04 distribution, running on a virtual machine with 4 vCPU and 8Gb RAM. The event2ledger module is then built as a containerized application, using a script file (Dockerfile) to ensure that all dependencies are already installed. The Hyperledger Fabric blockchain, in turn, demands the installation of a set of prerequisites detailed in the official documentation (Hyperledger.Docs, 2021). Subsections 5.1 and 5.2 describe the event2ledger and the Hyperledger Fabric blockchain, respectively.

5.1 Event2ledger

The event2ledger is a containerized application developed in node.js, that can be implemented as a service in the Docker Swarm, and have three functions: it collects all the container lifecycle events in a non-intrusive way, generates the transactions containing the event collected, and send them to the blockchain API. It can also be implemented directly on Docker, through the docker run command. The app.js file contains the code necessary to connects to the Docker API, receive the events, generate and send the transactions to Hyperledger Fabric API. To perform these functions, it is necessary to provide the IP addresses of Docker API and Hyperledger API and also the JSON Web Token (JWT) used to connect to the Hyperledger API and send the transactions.

After the container creation and execution, the event2ledger application connects to Docker API endpoint "/events", which streams in real-time the container events generated. Then, for every event, it creates an array containing the received data and sends it to the blockchain in JSON format, through a POST request to Hyperledger Fabric API. The request head must contain a valid JSON Web Token (JWT) token, previously generated and associated with one of the actors' private keys, to be allowed to send the transaction. Its body, in turn, contains the chaincode function to be called, the endorsing peers' addresses, the chaincode name, and the event data.

5.2 Hyperledger Fabric Blockchain

The Hyperledger Fabric blockchain implemented in this paper is a consortium model, composed of three participants. Each participant has one peer, that is responsible for endorsing, validate, and store the received transactions. The blockchain has, also, three orderer peers, responsible for the transaction ordering and distribution, and for reaching the consensus through the Raft mechanism.

The Hyperledger Fabric transaction flow has three steps: execute, order, and validate. The process starts with a transaction proposal that triggers a specific chaincode function. In this model, the proposal is sent to some peers, and the client needs to collect a given number of endorsements before sending the transaction to the orderer nodes. The endorsing process consists of the execute phase, when the transactions are executed, signed, and sent back to the client. When the proposal has enough signatures to satisfy the endorsement policy, it is submitted to the orderer node, which orders the transactions and sends them to all blockchain nodes. In the last step of the process, all the peers validate the transactions, labeling them as valid or invalid, and update the ledger world state. Figure 3 presents a general overview of the Hyperledger Fabric blockchain implemented and the transaction flow.

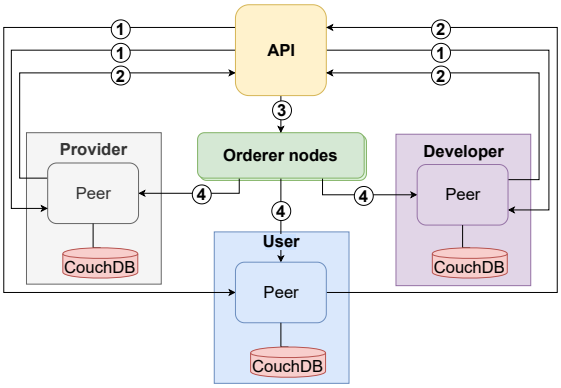

Figure 3: Proposed Hyperledger Blockchain model.

The Hyperledger Fabric API (Figure 3) receives the transaction sent by the event2ledger and sends it to the endorsing peers (1). The endorsing peers execute the transaction received, sign, and send it back to the API (2). The API collects the endorsed transactions until the endorsement policy is met. Then, it sends the endorsed transaction to the ordering nodes (3), which will order the transactions and broadcast them to all nodes of the blockchain (4). These nodes will validate the transaction and store its results on the ledger (CouchDB).

All the nodes that compound the Fabric blockchain, as the organization nodes, ordering nodes, and CouchDB instances are implemented as Docker containers. The proposed blockchain has three ordering nodes, three endorsing and committing peers responsible for endorse and validate the transactions, and three CouchDB instances, to store the world state, one for each committing peer. The implementation process creates a channel and joins the blockchain nodes to it. Then, the chaincode is installed on all endorsing nodes, containing a set of functions that allow to create and verify the events stored on the blockchain. The proposed solution uses a Hyperledger Fabric API to receive the transactions submitted by event2ledger. This API grants the endpoint security through a JWT token, that must be included on all calls. The API can also be implemented as a container, improving the solution scalability. The Table 3 presents the main configuration parameters applied to the blockchain.

Although it is possible for a node to belong to more than one channel, this model includes the installation of only one channel on each node. The

Table 3: Blockchain configuration parameters.

Parameter	Value
Number of channels	1
Chaincode language	Go
Ledger	Couchdb
Endorsement policy	Major Endorsement
Transactions per block	20
Block timeout	3s

chaincode used was developed in go, and has a set of functions used to register the event, retrieve data from all events and from a specific event. Other functions can be added as needed, through chaincode updates. The proposed solution uses CouchDB as the blockchain ledger, allowing the elaboration of complex queries. The endorsement policy adopted requires that the transactions be validated by the majority of participants. As for block configurations, the implemented model adopts blocks containing up to 20 transactions, with a timeout of 3 seconds (*i.e.*, after 3 seconds, the block is sent even if it does not contain 20 transactions).

6 TESTS AND RESULTS

The purpose of the test plan developed for this work is to verify that the events generated in the Docker environment are being properly collected and stored in the Hyperledger Fabric blockchain. The proposed test plan foresees the generation of a predefined number of container events, through a bash script. This script has a loop that is executed five times and creates and stops 20 containers, based on the hello-world Docker image. These containers are created with a specific name pattern, composed by *"IMG"+ n*, in which *n* is a number between 1 and 20. Each container created generates three events: create, connect and start. When the container is finalized, three events are also created: die, disconnect and destroy. So, the execution of the script must result in 600 container events that have to be collected by the event2ledger instance in execution. Each event collected must be equivalent to a transaction to be endorsed, ordered, and validated using the Raft consensus mechanism, and its result must be stored in the ledger. Thus, to validate the proposed test plan, it is necessary to identify the generated events in the blockchain.

To store the collected events on the blockchain, event2ledger uses a chaincode function named createEvent. This function, when executed, adds the event data onto the blockchain and creates a registry ID. This ID is a key composed of *EVENT + i*, where *i* is a sequential number. The ID aims to be useful to retrieve the events stored. Using a chaincode function named getAllEvents for this ID, it is possible to obtain all the events stored on the blockchain. The execution of this function should return 600 records, one for each event collected. In addition, transactions must have the signature of the nodes that endorsed them, and their validation code. Considering the endorsement policy applied in this work, the transactions need to be endorsed by the majority of the nodes, so it needs to have at least two endorser signatures. Also, it is possible to verify the event's origin using the transaction sender's key.

The script execution generated 600 events, of which 300 during the start phase, and 300 during the stop phase. The first point to be verified is if all the generated events were properly collected. To get access to blocks and transactions data, this work used Blockchain Explorer, a Hyperledger component that has a friendly interface and is useful to navigate on the blockchain and check blocks, transactions, metrics, and other information. The component dashboard shows that 51 blocks were created after the script execution, and the blockchain has 609 transactions, of which 9 transactions were generated during channel creation, node integration, and chaincode deployment. Using the getAllEvents chaincode function it was also possible to retrieve the events stored on the CouchDB ledger. This execution returned 600 events (EVENT0 - EVENT599), as expected previously. Then, the events were analyzed, to verify if their contents were collected and stored correctly. Code 1 brings an example of a collected event, stored on the ledger.

Code 1: Collected event.

```
{ "KEY": "EVENT135",
"RECORD": {
        "Action": "create",
        "ActorID": "d06ee64bf817b7f8d0d40f93b1cc",
        "EventID": "d06ee64bf817b7f8d0d40f93b1cc",
        "Image": "hello-world",
        "Type": "container",
        "From": "hello-world",
        "Name": "img4",
        "Scope": "local",
        "Status": "create",
        "Time": "1610736410",
        "TimeNano": "1610736410370693000",
}}
```

Code 1 allows to view the details of a collected event. The events are generated, collected, and stored in JSON format. Its contents show the event key, the action performed, actor and event identifiers, the image used, type of event, the container name, and other details as the timestamp. The analysis of the ledger

Table 4: Functional Requirements vs Solution Proposed.

Functional Requirement	Proposed solution	Details
FR1	Uses Docker API	Collect events using the Docker API endpoint /events
FR2	Uses JSON format	The collected events are sent in JSON format, via POST call, to the Hyperledger Fabric API
FR3	Meets via blockchain	Uses Raft mechanism to obtain consensus
FR4	Meets via blockchain	Each blockchain node has its own CouchDB node for storing records

content shows a total of 600 records for the generated events.

The second step of the test plan consists of checking the validity of the transactions. As described on 5.2, the valid transactions receives the validation code 0, and are labeled as "valid" at the end of the Hyperledger Fabric transaction flow. To review the validation status, it is necessary to verify the validation code field, in transaction details, as shown in Code 2.

Code 2: Transaction Details.

```
1  Transaction ID: 7de3eab67b3e9dad4d4a9c27c91ebe9b2c
2  Validation Code: 0 - VALID
3  Payload Proposal Hash:
     5e48bb6283965c2141561b46ae8535ef9e4
4  Creator MSP: ProviderMSP
5  Endoser: "ProviderMSP","DeveloperMSP","UserMSP"
6  Chaincode Name: eventdb
7  Type: ENDORSER_TRANSACTION
8  Time: 2021-01-15T18:42:31.871Z
9  Reads:
10 Writes:
```

The Code 2 shows details about a valid transaction, collected by Provider's event2ledger instance, as shown on Creator's MSP field. The Endorser field contains the signature of the transaction endorsers, which are all the blockchain participants (Provider, Developer, and User). Checking the validation code of the transactions on the ledger was possible to confirm that all was considered valid and had its results stored on the CouchDB ledger. Thus, the proposed solution meets the defined functional requirements (Table 4).

The FR1 is serviced using the Docker API, which streams the events in JSON format in real-time, ensuring a non-intrusive collection method. To send the collected event to the blockchain, the event2ledger uses a POST call that contains the JWT token in the head and the formatted transaction data in the body. To meet the FR3, the proposed solution adopts the Raft consensus mechanism, which not only handles consensus but also provides fault tolerance. To store the collected events, the proposed solution uses a CouchDB distributed ledger implementation.

6.1 Discussion

The event2ledger has been configured to collect events directly from the Docker Swarm master node, allowing the collection of all events generated by services, even if executed at other nodes. It was implemented as a single container to test its functionality but is also possible to implement the solution as a service, ensuring its scalability and fault tolerance. In the testing scenario implemented, the collection was performed by only one of the actors (in this case, the provider). To allow other actors to perform the collection, additional JWT tokens associated with each actor's private keys must be generated and inserted into the respective instances of the event2ledger. The test results showed that the events generated on the Docker environment were all collected and stored on the blockchain. However, it is possible to implement filters in the event2ledger so that only specific events are collected and sent to the blockchain, improving the application performance by focusing on the collection and validation process only on desired events. Despite the definition of the endorsement policy as "Majority endorsement", where validation is possible with 2/3 of the total nodes, the transactions were endorsed by all three nodes.

Although using the Hyperledger Fabric API facilitates integration, this layer adds complexity with JWT use and increases the solution latency. An alternative solution is to send the transaction using the Hyperledger Fabric peer binary. In this way, it would be possible to connect and send transactions directly to the blockchain, using the event2ledger private key, avoiding the need to create the JWT authentication token, and future improvement to the solution.

The number of orderer nodes defined is related to the Raft fault tolerance, given by $2f + 1$, where f represents the number of failed nodes. In this way, the adoption of 3 orderer nodes allows the fault tolerance of one node, without impacting the blockchain execution. The increase in the number of orderer nodes has the benefit of greater fault tolerance. However, it impacts an increase in network traffic, which may result in a reduction in performance.

7 CONSIDERATIONS & FUTURE WORK

The solution proposed was successful in collect and store the events generated and, with minor adjustments, can be implemented in environments that use other orchestrators, like Kubernetes. The storage of events on a blockchain made it possible to identify the event collector through the private key used in the transaction generation. Also, the distributed storage and cryptographic chaining characteristics ensure data availability and integrity.

It is important to note that the number of events generated in container virtualization environments can be high. Some blockchain configurations have a direct impact on its performance (e.g., channel composition, endorsement policy, block size, and block timeout) and must be adjusted according to the environment to be monitored. Benchmark tools like Hyperledger Caliper can be useful to verify the ideal blockchain configuration in terms of performance through transaction latency and throughput.

The Raft consensus mechanism allows an adequate security level combined with good performance, fitting most of the use cases where the participants are reliable. However, some environments may require a Byzantine fault-tolerant consensus mechanism. The Hyperledger Fabric has a modular architecture that allows replacement and customization of the consensus mechanism but does not yet have a Byzantine fault-tolerant consensus mechanism available.

As future works, we intend to develop a component that will allow fast log conflict identification between the actors regarding the collected events. To verify the throughput of the proposed solution we plan to execute a blockchain benchmark with Caliper, that can be useful to optimize the blockchain configuration. We also intend to implement some of the Byzantine fault-tolerant consensus mechanisms under development, such as SMaRt-BFT (Bessani et al., 2014), to verify the viability of the proposed solution in environments where actors are not reliable.

ACKNOWLEDGMENTS

The authors thank the support of FAPESC, and LabP2D / UDESC.
This work was supported by Ripple's University Blockchain Research Initiative (UBRI) and in part by the Brazilian National Council for Scientific and Technological Development (CNPq - grant 304643/2020-3).

REFERENCES

Bessani, A., Sousa, J., and Alchieri, E. (2014). State machine replication for the masses with BFT-SMART. In *44th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks*, pages 355–362.

cAdvisor (2020). cadvisor docs. https://github.com/google/cadvisor/.

Ciuffoletti, A. (2015). Automated deployment of a microservice-based monitoring infrastructure. *Procedia Computer Science*, 68:163–172. 1st Int. Conf. on Cloud Forward: From Distributed to Complete Computing.

Collectd (2020). Collectd docs. https://collectd.org/documentation.shtml.

Dawadi, B., Shakya, S., and Paudyal, R. (2017). Common: The real-time container and migration monitoring as a service in the cloud. *Journal of the Institute of Engineering*, 12:51.

Hykes, S. (2013). The future of Linux containers. Pycon US 2013. https://youtu.be/wW9CAH9nSLs.

Hyperledger.Docs (2021). Hyperledger fabric official documentation. https://hyperledger-fabric.readthedocs.io/en/release-2.2/prereqs.html.

Jamshidi, P., Pahl, C., Mendonça, N., Lewis, J., and Tilkov, S. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35.

Jiménez, L., Simón, M., Schelén, O., Kristiansson, J., Synnes, K., and Åhlund, C. (2015). Coma: Resource monitoring of docker containers. In *CLOSER*, pages 145–154.

Jo, H., Ha, J., and Jeong, M. (2018). Light-weight service lifecycle management for edge devices in I-IoT domain. In *Int. Conf. on Information and Communication Technology Convergence (ICTC)*, pages 1380–1382.

Laurie, B. (2014). Certificate transparency: Public, verifiable, append-only logs. *Queue*, 12(8):10—-19.

Newman, S. (2015). *Building Microservices*. OReilly, 1 edition.

Oliveira, F., Suneja, S., Nadgowda, S., Nagpurkar, P., and Isci, C. (2017). A cloud-native monitoring and analytics framework. Technical report, Technical Report RC25669, IBM Research.

Pourmajidi, W. and Miranskyy, A. (2018). Logchain: Blockchain-assisted log storage. In *IEEE 11th Int. Conf. on Cloud Computing (CLOUD)*, pages 978–982.

Rodriguez, M. A. and Buyya, R. (2019). Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience*, 49(5):698–719.