Scalable Computing: Practice and Experience, ISSN 1895-1767, http://www.scpe.org
© 2020 SCPE. Volume 21, Issues 4, pp. 709–725, DOI 10.12694:/scpe.v21i4.1838

DECENTRALIZED AND FAULT TOLERANT CLOUD SERVICE ORCHESTRATION

ADRIAN SPĂTARU*

Abstract. This paper proposes a decentralized framework for the orchestration of Cloud Services using heterogeneous resources residing in the homes of private individuals or small-scale clusters. The framework makes use of Ethereum Smart Contracts to provide a decentralized mechanism for discovering the different interfaces exposed by Cloud Components. The paper introduces a novel concept of Component Administration Networks, which are peer-to-peer networks that monitor and ensure the availability of the software components. The concept applied for the Orchestration process to ensure that the deployment of an Application continues in the presence of Orchestrator component failure. Checkpoints are used to address the continuity of the Management components, in general, and of the Orchestrator, in particular. In our proposal, checkpoint metadata is stored in a Smart Contract to assess the execution time of a Service to reimburse the participants that ensure its execution.

Key words: Blockchain, Decentralized Cloud, Service Orchestration

AMS subject classifications. 68M14,68M15

1. Introduction. Cloud Service Providers make use of warehouse-scale computers [1, 2] to provide Infrastructure and Services to businesses and entrepreneurs to accelerate the implementation and deployment of Cloud Applications. The advancement of the Internet of Things field has introduced new challenges concerning the bandwidth required for massive data transfers between the originator (residing at the end of the network) and processing services (running in the Cloud data centres). Fog Computing tackles the movement of Cloud Services closer to the data source. Such systems are generally hierarchical, each level processing data from an underlying level and sending it to an upper level, reaching the Cloud at its top. This reduces the amount of bandwidth necessary for transferring the data. Nevertheless, the cost of implementing Fog Networks is high in terms of both investment and maintenance.

We reckon that peer to peer networks of personal computers and small-scale private Clouds can participate in a Decentralized Cloud Platform, able to provide the resources required to execute a Cloud Service closer to the Service consumers. For example, a 3D artist can run a rendering application, as a Service, on a computer from his/her immediate vicinity. More complex services can be delivered to an area of consumers close to a local Cloud, and services executing on multiple local Clouds can discover and synchronize with each other. A vast amount of investigation has been pursued with respect to peer to peer systems consensus: [3, 4, 5, 6, 7], file sharing services such as IPFS [8], Kademlia [9], BitTorrent [10], and volunteer computing applications such as BOINC [11], XtremWeb [12] as well as dealing with saboteur nodes [13].

Blockchain technologies, such as Bitcoin [14] and Ethereum [15], have increased the number of peers willing to participate in globally distributed networks of computers, providing an economic incentive to maintain and extend a global replicated state machine. The Ethereum Virtual Machine makes transitions using a quasi-Turing complete instruction set, which can be used to define Smart Contracts comprising of arbitrary code. In order to limit the misuse of the platform, each instruction has a cost expressed in gas, out of which the data-store instruction has the highest cost. This renders unfeasible any attempt to use the Blockchain as a Storage Service but is a guard against infinite execution. Instead, Smart Contracts are intended to contain just the business logic of an application. The state machine is updated by transactions which are organized into blocks on a Blockchain. This in turn requires each node participating in the network to execute all state updates from the transactions in a block locally, which hinders the performance of the system. The advantage

*Department of Computer Science, West University of Timişoara. (adrian.spataru@e-uvt.ro). Questions, comments, or corrections to this document may be directed to this email address.

is that each node can read from the state machine locally. At the time of this research, a number of 1, 382, 198 nodes[1] have been historically registered with the network, out of which at least 10, 000 nodes (the maximum shown by the Etherscan interface) are active daily.

Several decentralized platforms have been developed to make use of the Ethereum Blockchain and the power of Smart Contracts. FileCoin [16] offers a peer-to-peer storage service based on IPFS, using Ethereum for payments and access management. Data storage is verified using a Proof of Replication [17] mechanism which validates that a node has dedicated space to hold file blocks under a given replication requirement. Other Blockchain solutions have been investigated in the direction of Autonomous Vehicles cooperation [18, 19] and Internet of Things cooperation [20, 21]. By using a blockchain, the decentralized platform offers trust guarantees concerning the correctness of the Smart Contract execution. Additionally, each state transition can be audited, allowing for a transparent decision-making process.

Nevertheless, peer-to-peer systems (especially networks formed by personal computers) have always suffered from a lack of predictability with respect to the availability of the participating nodes. This is not a limiting factor in the case of a Blockchain system, where nodes are used only to store the blocks of transactions. However, when a node is assigned to run a Cloud Service, then this node should remain available for the execution of this Service. In case of failure, high-availability of the Service can be ensured using replication, yet a more desirable solution is to also restart the failed Service. Moreover, if the system is envisioned to be fully decentralized, then high-availability and fault tolerance should also be ensured for the Cloud Management Components.

This paper enhances an existing architecture (developed in the framework of the CloudLightning Project [22]) to allow for the decentralized management of a public Cloud platform that aggregates resources owned by private individuals. The enhanced platform ensures the Continuity of Cloud Applications in presence of Service Failures and Continuity of the Cloud Platform in presence of Management Component failures. Moreover, we provide mechanisms to ensure that a fair price is calculated based on the time that a node has dedicated to executing a Cloud Service. In summary, this paper provides the following key contributions:

- an architecture that allows for the decentralization of the Components of the Cloud Platform and thus the resource registration and assignment mechanisms, using Smart Contracts
- the concept of Component Administration Networks together with corresponding protocols; these networks provide a bridge from the Smart Contract World and the Software World and ensure the progress and continuity of Management Components by assigning them work, saving checkpoints, and monitoring their availability.
- a mechanism allowing for fault-tolerant Application Orchestration which makes use of the Component Administration Networks; this mechanism ensures that the Services composing an application can continue to be deployed in the presence of an Orchestrator failure and the user is taxed only for the amount of time a Service has executed;

The rest of the paper is structured as follows. Section 2 recalls the CloudLightning Architecture and presents its main features. Section 3, presents our architecture that offers decentralized management. Section 4 defines the concept of Component Administration Networks and investigates its behaviour through simulations of different failure types and rates. In Section 5 we use this concept to define the Application Orchestration mechanism which tracks the execution and computes the price based on the observed execution. At last, conclusions are presented in Section 7.

2. CloudLightning Architecture. Figure 2.1 highlights the principal components of the CloudLightning Architecture focusing on the three layers.

The **Gateway Service** offers user-facing functions such as maintaining a catalogue of Services that can be composed into an Application using an intuitive **User Interface** (UI). Services and Applications are defined using the Topology and Orchestration Specification for Cloud Applications (TOSCA). CloudLightning Services improve the flexibility of the Cloud User by allowing him to describe an Application composed of Abstract Services which have multiple hardware-specific implementations. The **Service Optimization Engine** is a component that reads this description and contacts the Resource Manager Layer to choose the best-suited implementation for each Abstract Service, depending on the requirements set by the user (performance, cost)

[1] https://etherscan.io/nodetracker/nodes, accessed 16 Nov 2020

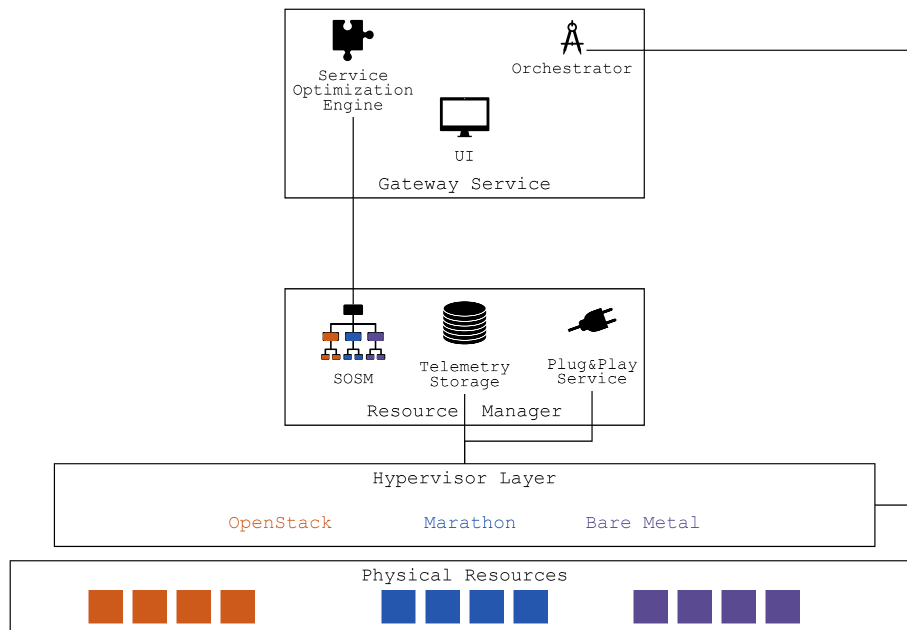


Fig. 2.1. *CloudLightning Components Architecture*

and the availability of the System. This mechanism also improves the flexibility of the Cloud Provider, by allowing him to offer a more efficient resource type like Graphical Processing Units (GPUs) or Many Integrated Cores (MIC) cards, in order to reduce its energy costs. Once the resources are selected, the Abstract Services are replaced by their Concrete Implementation in the Application definition. Finally, the **Orchestrator** component reads a Topology composed of Concrete Services and deploys it, taking into account any dependencies. It then monitors the deployment and execution of the Services, being able to restart a Service in case of failure.

On the bottom layer, physical resources are under the control of a **Hypervisor**, achieving different types of abstractions: Virtual Machines or Containers. The Hypervisor head node needs to contact the Plug&Play component to register with the **Self Organizing Self Managing** (SOSM) Resource Management System. Additionally, a telemetry client must be installed to send data to the Telemetry Service (a time-series database). The Hypervisor APIs are used to create VM and Container Images and to start Instances. Bare Metal servers can also be managed by the creation of accounts when this type of resource is selected. Further configuration and execution are managed using bash scripts.

The SOSM System [23] is a hierarchical resource management system that aggregates monitoring information collected at the bottom level and assesses the efficiency and performance of resources based on some specific business objectives (e.g., energy efficiency, computational performance). The assessment functions are weighted based on some importance for each aspect and propagated up the hierarchy. Each component of the hierarchy is described by a Suitability Index (with respect to the desired state of the system) and can be used to guide the Service requests to reach resources that are more energy or computationally efficient. The system behaviour has been investigated using Large Scale Simulations [24] which show that this system is able to optimize the resource utilization and performance metrics of data centres of over $100,000$ servers.

A self-healing implementation of the SOSM system has been presented in [25]. Each component is replicated and has either the role of a controller or a replica. The controller ensures enough replicas are available to cope with the current load of the system. A Message Queuing System is employed to route the messages between the different layers of the systems, ensuring only one replica is processing a request for resources. The replicas broadcast heartbeat messages to all corresponding replicas and construct a list with their startup time and status. A leader failure is detected if enough time has passed until a heartbeat was received. Then, the eldest replica will take the role of the controller and inform the rest.

The Orchestrator can detect the failure of a Cloud Service and issue the restart process. If any Services were dependent on the failed Service, then they should be reconfigured with the new properties (i.e., endpoints). Virtual Machines and Containers should use data volumes, to avoid losing progress if restarted. The Client can ensure the high availability and load balancing of its Application by requesting multiple instances of the same Service.

3. Decentralization of Cloud Components. In a previous paper, [26], we have proposed mechanisms for Decentralized Scheduling of Cloud Services using Smart Contracts and investigated the operational constraints and cost of such a system. Our investigation has revealed a high transaction cost when encoding the scheduling optimization algorithm in the Smart Contract. Moreover, the high cost reduces the number of transactions that can fit in a block, decreasing the transaction throughput and thus increasing latency for sealing a resource assignment.

A more efficient solution is to allow the Cloud Users to read the state of the Smart Contract and apply the resource selection algorithm locally. Once a resource is selected, a function is called in the Smart Contract to seal this assignment. Our investigation on a real-world Cloud scenario has shown that the cost for this method is 1/5 of the previously mentioned cost. Unfortunately, several users can select the same resource at the same time, which leads to conflicts and thus rejected transactions. Overall, this leads to the latency of the two methods to be comparable. This result leads to the necessity for a component that synchronizes the decisions made by users, such that no conflicting transactions are sent to the Smart Contract.

The modular architecture of the CloudLightning system allows us to decouple several components, paving the way to the decentralization of control and fault tolerance of the management components. Figure 3.1 depicts our proposed architecture. A public Blockchain able to execute Smart Contracts is used to synchronize the knowledge about the available components.

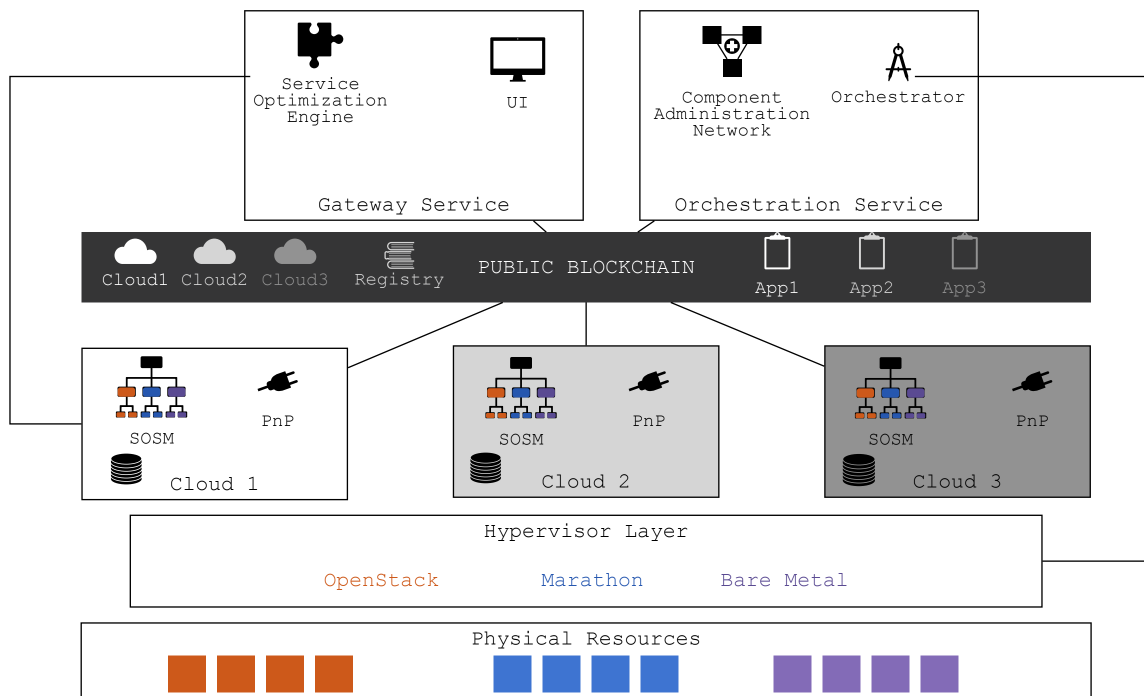


Fig. 3.1. *Augmented Decentralized Architecture*

In order to allow for the discovery of the decentralized components, we use three Smart Contracts:

1. **Registry Contract** – the main contract for the system. It is used to register Cloud Providers, Services and Applications. Additionally, it is used to maintain the contact information and availability of the

nodes taking part in the Component Administration Network.
2. **Cloud Contract** – holds resources descriptions and price, as well as the endpoints for the Scheduler and Plug&Play components.
3. **Application Contract** – created for each Application in order to track deployment status and payments. Checkpoints metadata is stored in this contract, which serves as proof for computing payment to the entities involved in the execution.

In this augmented architecture, the Gateway Service is a client-side application. It can read the state of the Registry Contract from local storage (if the client also runs an Ethereum node) or using a remote REST call (if the client only runs the Gateway Service). From the Registry state, the client can read the available Service and Applications definitions, as well as the registered Clouds. The Service Optimization Engine is able to reach the Cloud Scheduler endpoint in order to ask for resources, by reading the information present in the Cloud Contract selected by the client. A Cloud Contract can be backed by a SOSM Resource Management System or any other Scheduler that implements the Blueprint-based Resource Discovery Protocol described in [28, Ch. 4.1]. The Gateway Service is thus decentralized and multiple instances of this Component are synchronized through the Registry Contract and Cloud Contracts.

The Orchestration process is decoupled from the Gateway. This process is distributed over several nodes that are coordinated by a Component Administration Network (CAN). Instead of mining, a subset of the Blockchain-participating nodes can be part of a CAN and be rewarded for monitoring and storing Management Components states. Component Administration Networks are further detailed in Section 4.

Resource owners are expected to use a thin client that allows them to read the content of the Registry Contract and learn about any available Cloud Contracts. The owner can then contact the Plug&Play Service which, after validation, registers the physical resources with the SOSM system and updates the Cloud Contract.

3.1. System Initialization. An Ethereum Blockchain Network is assumed to work as an external component to our proposed system. To achieve better latency and not suffer from the Ether crypto-currency volatility, we reckon the best practice is to create a new network, used only for this purpose. The system initiator can dedicate some nodes to bootstrap this chain and extend the public ledger. Additionally, the system initiator must dedicate some nodes to bootstrap the Orchestration Component Administration Network (OCAN).

The following steps are required to initialize the system:

1. Deployment of Registry Contract. This is initialized with the address of the owner, an empty list of OCAN candidates and a null value assigned for the OCAN leader.
2. Component Administration Network bootstrap:
   i) Registration of nodes as CAN candidates. Each node is associated with an Ethereum account
   ii) The system initiator selects a node from the CAN candidates to be the leader of the OCAN
   iii) The selected OCAN leader inserts its contact information (e.g., IP, port) in the Registry contract
   iv) Candidate nodes learn the contact information of the leader and contact it to join the OCAN.
   v) Candidate nodes get verified as Orchestration nodes by the OCAN leader.
3. Services and Applications are registered in the Registry Contract using the TOSCA specification.
4. A Cloud Service Provider calls the Registry function to deploy a new Cloud Contract, providing the endpoints of the Scheduler and Plug&Play components. Additionally, a Provider should also provide the public key of the Scheduler, used to sign the reservation for resources to a client.
5. Resources can be registered with the Cloud Contract:
   i) The resource owner contacts the Plug&Play Service defined by a Cloud Contract which he/she wishes to join.
   ii) The Plug and Play Service verifies the Hypervisor API and telemetry endpoints and decides to accept or reject
   iii) If accepted, the resource is added to the management system and registered in the Cloud Contract. Note that resources do not need to run an Ethereum node. They are registered and removed from the Cloud Contract memory by the Plug&Play component.

Steps 2, 3, and 4 do not depend on one another and can be executed in parallel. The resource manager is updated by the Hypervisors with telemetry data periodically. If the data cannot be obtained from the registered resources, it can inform the Plug&Play Service to deregister it.

3.2. Application Deployment. An Application is defined using the TOSCA Topology specification. The Client creates an Application definition using the UI and registers it in the Registry Contract before deployment. Alternatively, Application definitions registered by others can be referenced by id.

An application is deployed in the following steps:

1. The User selects an Application and a Cloud. A Blueprint is generated by the Service Optimization Engine and sent to the Scheduler for assigning the resources and choosing the concrete implementation in case of Abstract Services. In the successful case, the Scheduler will reserve the resources for a given amount of time for this Application. The user is returned a signed message encompassing this reservation. The user can accept the implementation and selected resources, or he/she can ask for their release (such that the resources can be reserved to someone else before the expiration of this reservation).
2. To instantiate the Application, the user calls the corresponding function in the Cloud Contract, providing the signed message from the Scheduler. The Application Contract is deployed if and only if the reservation has not expired and none of the resources has been deregistered in the meantime.
3. The OCAN leader is informed that a new Application Contract has been deployed. This step can be achieved by the Gateway or by the use of Ethereum events, with the OCAN leader listening for new Application Contracts.
4. An Orchestrator is selected by the OCAN leader to be in charge of the Application deployment; this decision is sent to the OCAN nodes to be recorded on the ledger.
5. The Gateway is now able to query any OCAN node to learn about the managing Orchestrator. This Orchestrator is contacted for information about the runtime properties of the Services under deployment.

Several failures can occur throughout the lifetime of the proposed system. *Node failures* refer to the state when a node becomes unavailable, either due to hardware problems or network disruptions. This can, in turn, generate *Management Component failures* if the node is running a Management Component or *Service Failures* if the node is a physical resource executing a Service. Our system cannot replace a failed hardware component or resolve networking disruptions but mitigates their effect by enforcing fault-tolerance at the Management Component level and Application deployment continuity at the Orchestration level.

4. Component Administration Networks. A Component Administration Network has a two-fold purpose. First, it bridges together the land of Smart Contracts with the land of Software Components. Second, it provides the means for monitoring and enforcing a set of replicas in order to tolerate faults. The network of nodes implements a replicated state machine which has two functions. First, it maintains a ledger of transactions related to the network of nodes and the supervised components, different than the Ethereum ledger. Second, the nodes execute a replicated file system to store data associated with the supervised components.

Figure 4.1 presents the layered architecture of this proposal. On the bottom layer, there is the peer to peer network that collaborates for maintaining the Ethereum Blockchain, where the Registry Contract is deployed. Some of these nodes can be part of the Component Administration Network.

The middle layer is concerned with operations for managing the nodes. This is the first layer where we identify the leader of the network, which is responsible for ordering and validating all transactions related to the CAN. The replica nodes will accept any state update from the leader. For this layer we propose the following protocols:

- Join – protocol for a new node to join the network
- RemoveNode – protocol for removing a node that has been discovered to be faulty
- LeadElect – protocol for electing a new leader if the current one has been discovered to be faulty

The top layer is concerned with the administration of Components. Again, the CAN leader is responsible for ordering and validating state updates at this level. This layer is concerned with the following protocols:

- Register – a component gets registered with the network
- Deregister – a component has been unresponsive and is removed
- AssignWork – a component is assigned work
- CheckpointWork – a component requests the network to store a Checkpoint
- ReassignWork – a component has been removed and work is reassigned to another

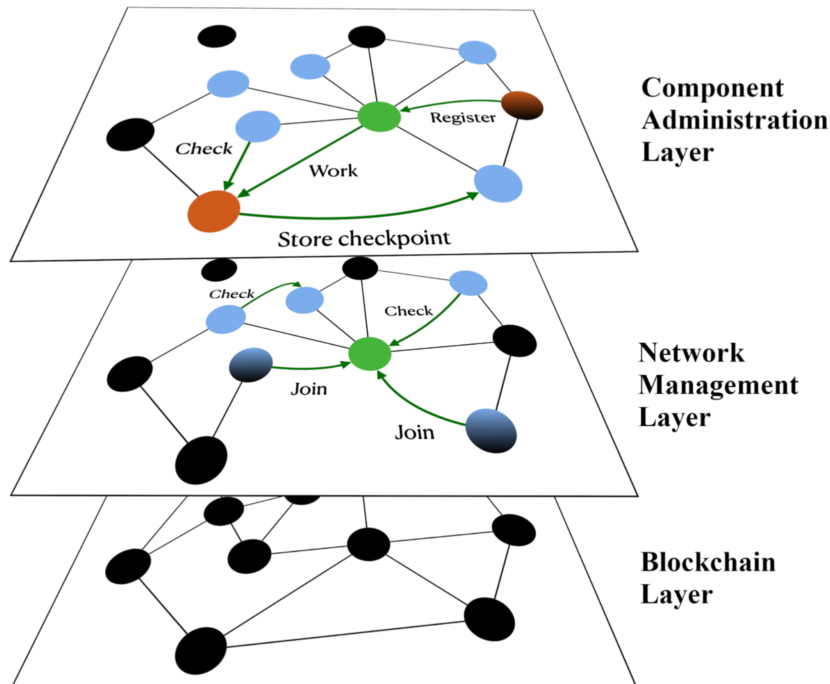


Fig. 4.1. *Layered architecture of a Component Administration Network*

- FinishWork – a component is requested to terminate the execution of a given piece of work (termination of a Cloud Application).

4.1. Network Management Protocols. The network makes use of the Registry Contract to know who is the current leader and what nodes are registered with the network, in order to reward them for their work. Nodes that want to participate in this network must first register in the Registry Contract as candidate nodes. When the system is initialized, the Registry owner selects one of the candidate nodes as the leader of the Network. When the contact information of the leader is known, other candidates can proceed to join the network by contacting the CAN leader.

4.1.1. Joining a network. The protocol for joining a network is presented in Algorithm 1. The Registry owner must set N_{min}, a minimum number of replicas required for the network. The Registry owner must also set N_{max} the maximum number of nodes that can be part of the network. The purpose of N_{max} is to limit the slowdown of the network due to synchronization and data transfers. The *read* function represents a query to the Smart Contract to read specific properties.

A candidate node will attempt to join the network if the size of the network is less than its maximum size. Any candidate node that contacts the leader will be accepted until the minimum number of replicas is reached. After this point, the leader will accept new replicas only if the *churn rate* is greater than the current surplus of nodes (network size minus N_{min}). The churn rate is the difference between nodes that left the network and nodes that joined the network. This is an adaptive mechanism that allows the network to grow at the same pace as nodes are crashing. If a node is accepted, then this decision is broadcast across the nodes and the node's contact information is registered in the Registry Contract.

We analyze the dynamics of this network in terms of its size with different properties. We design an experimental simulation consisting of 100 steps, executing the following at each step:

1. A candidate node will request to join the network with probability $j \in \{.2, .1, .05, .025, .01\}$
2. A CAN node will crash with probability $f \in \{.5, .75\}$. Nodes joining in this step can also crash in this same step.

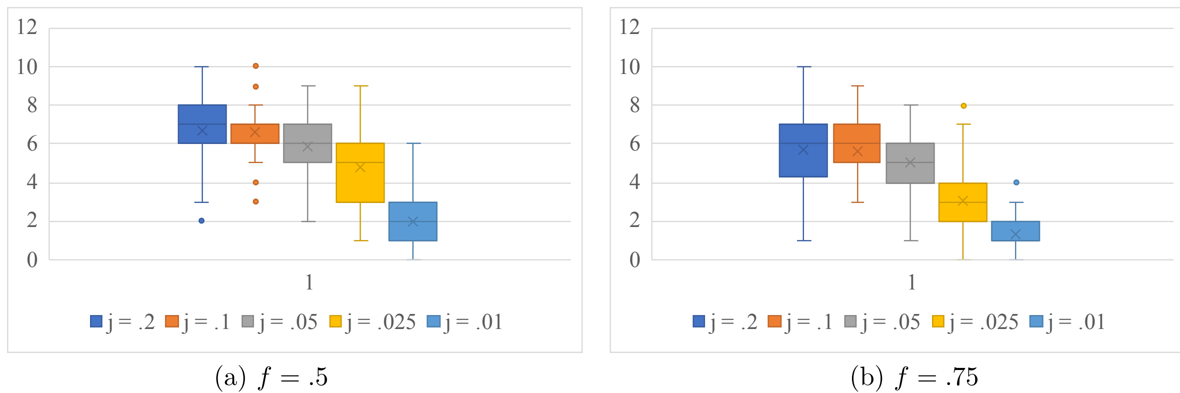
First we consider a network with $N_{min} = 4$, $N_{max} = 10$. We consider $N_{total} = 100$, the total number of

Algorithm 1 Protocol for joining CAN

```
1: current ← read(Registry, CAN.size)
2: max ← read(Registry, CAN.max)
3: leader ← read(Registry, CAN.leader)
1: function askJoin
2:     if current < max then
3:         sendJoinRequest(leader, contactInfo)
4:     end if
5: end function
1: function listenJoin(contactInfo)
2:     if current = max or checkAvailable(contactInfo) = false then
3:         return reject
4:     else
5:         if current > min then
6:             if current − min > churn then
7:                 return reject
8:             end if
9:         end if
10:    end if
11:    broadcast_node_join(peers, props)
12:    registerPeer(SC, props)
13:    return accept
14: end function
```

participating nodes and $C = N_{total} - N_{current}$, the number of candidate nodes. When the network is started, the leader is appointed by the Registry owner.

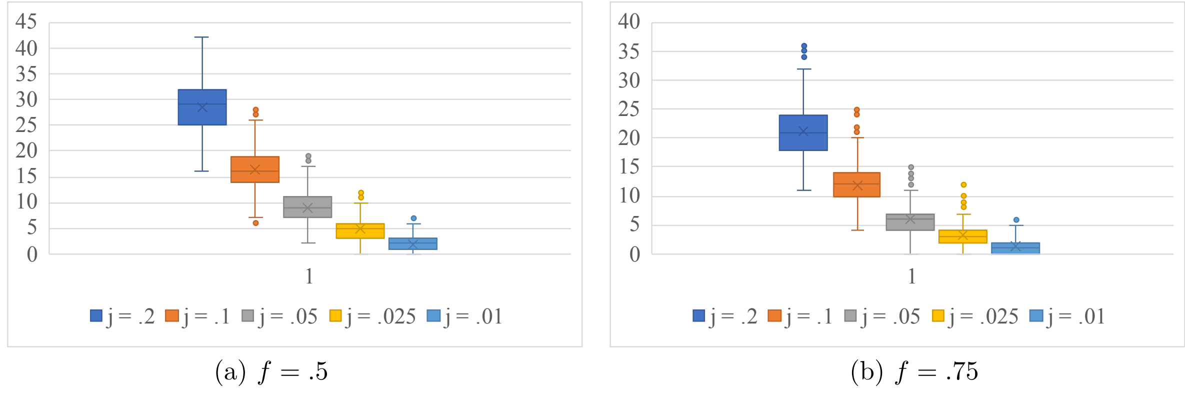
Figure 4.2 shows the size of the network using box plots. Figure 4.2.a represents the size dynamic when $f = .5$, while Figure 4.2.b represents the size dynamic when $f = .75$. For $f = .5$, results indicate that the joining probability of the nodes should be at least 0.025 in each step in order to have, on average, a network of the minimum desired size. If $j = 0.01$ the average size of the network is 2, reaching states where the size is 0. All other joining probabilities reach states where the size of the network is smaller than the minimum desired size. For $f = .75$, if $j \in .01, .025$ then the failure of the network is guaranteed. Other joining probabilities have an average size larger than the minimum but still reach states where the size is smaller than the minimum.



(a) $f = .5$ (b) $f = .75$

Fig. 4.2. *Size of a CAN for different join and failure probabilities for* $N_{max} = 10$

We repeat the experimental simulation, with N_{max} set to 100, with the results presented in Figure 4.3. Although the size of 100 is never reached, the network benefits by having a greater tolerance to node failures.



(a) $f = .5$ (b) $f = .75$

FIG. 4.3. *Size of a CAN for different join and failure probabilities for* $N_{max} = 100$

For $f = .5$, we observe that the candidate nodes must be willing to join with probability $j > 0.1$ in order for the network to maintain a size greater than the minimum desired. For $f = .75$, results are similar. Given the simulation results, Algorithm 1 maintains the network operational if a large enough limit, N_{max}, is set and if sufficient new nodes are willing to join the network. In our simulation observed that for $N_{max} = 100$ and $j = .1$ the network is able to maintain a minimum desired size in presence of failures with probability $f = .75$, although we do not expect probabilities of this size in practice.

4.1.2. Node Removal. We propose a mechanism for checking the state of the nodes in the network. All nodes, with the exception of the leader, will execute the protocol described by function *checkState* in Algorithm 2. A node randomly selects another node of the network, including the leader, to check its status. If the node responds, then the function stops. If the node does not respond within a given timeout, then this node is considered in a failed state. If the node is a leader, then a Leader Election process is started; this process is described in the following subsection. If the node is a normal replica, then a request to remove the node is sent to the network leader.

Algorithm 2 Protocol for removing a CAN node

1: **procedure** CHECKSTATE
2: $node \leftarrow randomSelection(peers)$
3: $status \leftarrow ping(node)$
4: **if** $status \neq OK$ **then**
5: **if** $node \neq leader$ **then**
6: $sendRemoveRequest(leader, node)$
7: **else**
8: $leadElect(peers, self)$
9: **end if**
10: **end if**
11: **end procedure**

1: **procedure** LISTENREMOVE(R)
2: $v \leftarrow collectLeaveVotes(R)$
3: **if** $count(v, accept) > \frac{current}{2}$ **then**
4: $broadcast_node_leave(peers, R)$
5: $deregisterPeer(Registry, R)$
6: **end if**
7: **end procedure**

When receiving a Removal Request for node R, the leader holds a vote to remove this node. This step

is necessary because the proposer node might be the only one seeing R as unavailable (e.g., the proposer experiences a network disruption). If a simple majority of the network considers R to be unavailable, then this node is removed from the network. This decision is broadcast to the network and the contact information for node R is removed from the Registry Contract.

4.1.3. Leader failure and election. A leader failure is detected by the periodical execution of function *checkState*. Algorithm 3 presents the protocol for being selected as the leader. We use the terminology from the Paxos Algorithm [27] and consider the nodes which detect the failure of the leader as *Proposers*. The remaining nodes behave like *Acceptors*. Acceptors run the protocol described by Algorithm 4.

Algorithm 3 Procedure for CAN Leader Election

1: **procedure** LEADELECT
2: $n \leftarrow read(SC, leaderNonce) + 1$
3: $votes \leftarrow map(collectLeadPromise(peers, n, self))$
4: **if** $count(v.values, promise) > \frac{current}{2}$ **then**
5: $acc \leftarrow [\ \ ]$
6: **for** p in $peers$ **do**
7: **if** $v[p] = promise$ **then**
8: $v \leftarrow sendLeadAccept(p, n, self)$
9: $acc.append(v)$
10: **end if**
11: **end for**
12: **if** $count(acc, accept) > \frac{current}{2}$ **then**
13: $receipt \leftarrow registerLeader(SC, n, acc, self)$
14: $broadcast_leader_change(peers, receipt)$
15: **end if**
16: **end if**
17: **end procedure**

A *nonce* value is maintained in the Registry Contract to be associated with leader election processes. This value is read by a Proposer, which increases its value by 1 and attempts to collect *promises* from its peers. If multiple Proposers exist, then all of them will have the same nonce value. If a Proposer receives a simple majority of promises, it proceeds to the next step. In this step, it will ask all promising nodes for their accept vote. If a simple majority of final votes is gathered, then this node is the new leader. The new leader must update the Registry Contract, providing the signed accept messages. The Smart Contract checks the signatures, and if all are valid then it updates the nonce value and the leader contact information. The new leader broadcasts a receipt of this update to the nodes, which can read the new state of the Registry.

Algorithm 4 Protocol for promising to a new CAN Leader

1: **function** LISTENLEADELECT$(n, node)$
2: $nonce \leftarrow max(nonce, read(Registry, leaderNonce))$
3: **if** $n \leq nonce$ **then**
4: **return** *reject*
5: **end if**
6: **if** $ping(leader) \neq OK$ **then**
7: $nonce \leftarrow n$
8: **return** *promise*
9: **end if**
10: **return** *reject*
11: **end function**

When receiving a *Promise* request from a peer, a node will first compare the local nonce value with the

one written in the Registry Contract and will update its local value with the maximum between the two. If the proposed nonce, n, is not greater than local nonce, then the promise is rejected. If the leader is indeed unavailable then the local nonce is updated and a promise is returned. When receiving an *Accept* request, the node will return a signed Accept message only if it has already promised to the requester using this nonce.

The reason this protocol takes place outside the Registry Contract is to limit the number of Ethereum transaction required to elect a new leader and thus reducing the recovery time for the CAN. If the protocol took place in the Smart Contract, a minimum number of $n/2 + 1$ Smart Contract calls need to be issued, one for each vote. In our proposed protocol the votes are cast at the Network Management Layer and only one function call is made to the Smart Contract.

4.2. Component Administration protocols. Components register with a CAN by contacting the network leader providing an endpoint where the software is reachable. If the leader is able to access the endpoint then it decides to register this component and broadcasts a RegisterCompoenent(compID) transaction to update the CAN ledger and inform the network about the new Component which needs monitoring.

The failure of a Component is detected using the *checkState* function. Besides checking the state of a randomly selected peer, a node will also check the state of a randomly selected Component. If a Component is considered unavailable, then the node will ask the leader to remove this Component. The leader proceeds by holding a vote, similar to the vote for removing a node. If a component had any work assigned, this will have to be reassigned to another component, together with all the checkpoints that the failed Component was able to generate.

4.2.1. Work assignment. In general, a Work Entry is characterized by an identifier and a description. In our case, the identifier is the address of an Application Contract, and the description is a TOSCA Topology. A client can inform the CAN leader about a work entry, which in turn will select a component to execute this work, for example using the Round Robin method. This is achieved in two steps:

1. The leader contacts the selected component which validates the work entry and returns an Accept message.
2. The leader will update the CAN ledger with a *AssignWork(workID, compID)* transaction, where *workID* is the Work Entry identifier and *compID* is the identifier of the component

A component can save checkpoints during the execution of the Work Entry. This is done by contacting the leader to store the checkpoint information (identified by the digest of the data) on the CAN storage. This is done in two steps:

1. The leader will select a peer as the initial storage node for this checkpoint and return its contact information to the component. The component can start uploading the checkpoint information on this initial node.
2. The leader issues a *CheckpointWork(workID, compID, dig(cp))* transaction, which tells the peers to start replication for the checkpoint identified by digest $dig(cp)$.

In the event of a Component removal because of failure, the leader will reassign any Work Entries to new Components. After the selection of a new Component which validates and accepts the corresponding Work Entry, the leader updates the CAN ledger by broadcasting a *ReassignWork(workID, compID, cpa)* transaction to the network. The newly assigned component can retrieve the data corresponding to the digests in the checkpoint array, *cpa*, from any of the CAN nodes.

A Work Entry can be stopped and its checkpoint information can be removed from the replicated storage system. At first, the leader will inform the assigned Component to execute any shut-down instructions encapsulated in the Work Entry description; a signal will be returned when finished. Finally, the leader issues a *FinishWork(workID)* transaction which informs the peers to remove any checkpoint entries associated with the given Work Entry id.

5. Fault tolerant Orchestration. Given the protocols described by the previous section, we use a CAN to manage the Orchestration process. Orchestrators are registered with the Orchestration Component Administration Network (OCAN) as Components, which execute Work Entries described by Application Contracts.

The leader of this network adds the checkpoints metadata to the Application Contract. A checkpoint metadata contains the checkpoint type, a timestamp and an issuer. For the Orchestration process we consider

4 checkpoint types:

- *SERVICE_UP*,
- *SERVICE_DOWN*,
- *APP_OK*,
- *APP_SHUTDOWN*.

These checkpoints are used to compute payment duties to the different entities involved in the Application execution. The leader is responsible to periodically call an Application Contract function that computes the payment and sets funds at the disposal of the entitled entities.

An Orchestrator selected by the leader proceeds with reading the Application Topology and selected infrastructure and starts the deployment of the Services, using the corresponding Hypervisor API. After each successful deployment, the Orchestrator Component will request the storage of a *SERVICE_UP* checkpoint, containing information about the deployment and its monitoring id issued by the Hypervisor. This information is useful to an Orchestrator replica to continue the deployment in case the current Orchestrator fails.

A Service failure can be detected by the Orchestrator during the periodical checking of the state for each deployed Service. If a Service failure is detected, a *SERVICE_DOWN* checkpoint is made to acknowledge the new state and redeployment is attempted. If the Service cannot be redeployed after a given number of retries set in the Application Contract, then the whole Application is shutdown, issuing a *APP_SHUTDOWN* checkpoint. Periodical checkpoints of type *APP_OK* are issued at intervals set in the Application Contract. These checkpoints are issued only if all Services are executing.

In Figure 5.1 we present a sequence diagram that illustrates the continuity of the deployment process in the presence of Orchestrator component failures.

We consider the case of the Ray Tracing Application developed in the CloudLightning Project, which is composed of two Cloud Services, a back-end rendering engine and front-end Web Service depending on the former for high fidelity object rendering. Before the events presented in the sequence diagram, we consider the client has contacted a Cloud Contract for the creation of an Application Contract, using the steps described by Section 3.2. The OCAN leader is contacted by the client (via the Gateway User Interface) to start the deployment of this Application. The OCAN leader selects *Orchestrator 1*, and after work validation it issues an *AssignWork* transaction. The Orchestrator deploys the first Service and stores a *SERVICE_UP* checkpoint. During the periodical monitoring, an OCAN node observes that this Orchestrator has crashed. The leader removes this Orchestrator from the list of managed components and reassigns the work with one checkpoint to Orchestrator 2, which continues the deployment with the second Service, and issues another *SERVICE_UP* checkpoint. If during the periodical inspection a Service has failed, the *SERVICE_DOWN* checkpoint is issued for this Service. Since all Services are available, an *APP_OK* checkpoint is issued.

5.1. Payment. Several entities are ensuring the execution of an Application: The Cloud and its resources, the OCAN nodes, and the Orchestrators; all of them need to be reimbursed. If an Application runs for a long time and a large number of checkpoints are generated, a transaction computing the total payment might run out of gas. Nevertheless, the granularity of checkpoints increases the fairness of the price the client pays. Therefore, we propose a method for interim payments, described by Algorithm 5.

All checkpoints registered by the OCAN leader are stored in the checkpoint array, *cp*. The last paid checkpoint index is stored in variable *lp*, initialized with -1 during Application Contract creation. Payments are accumulated during the life-cycle of an Application in a map represented by the *salary* variable. This approach solves two problems: on one hand, the OCAN leader (which calls the interim payments function) does not need to pay a fee for transferring the funds to the other entities; on the other hand, all entities can decide to withdraw the funds they earned at the end of the execution and only pay the withdraw fee once. The withdraw fee is the base price for executing a funds transfer transaction on the Ethereum Blockchain. The start time (in Unix timestamps) for each service is stored using a map represented by variable *start*.

The function *InterimPayment(k)* in the Application Contract, where k represents the number of checkpoints to be paid can be simulated locally to make sure a large enough value can be set for k (such that fewer transactions are made) without exceeding the gas limit. When called, the variable *newlp* is initialized with the last payment index and is increased for each paid checkpoint, and the variable *total* is set to 0 and is used to check if the total payment duties exceed the currently available funds.

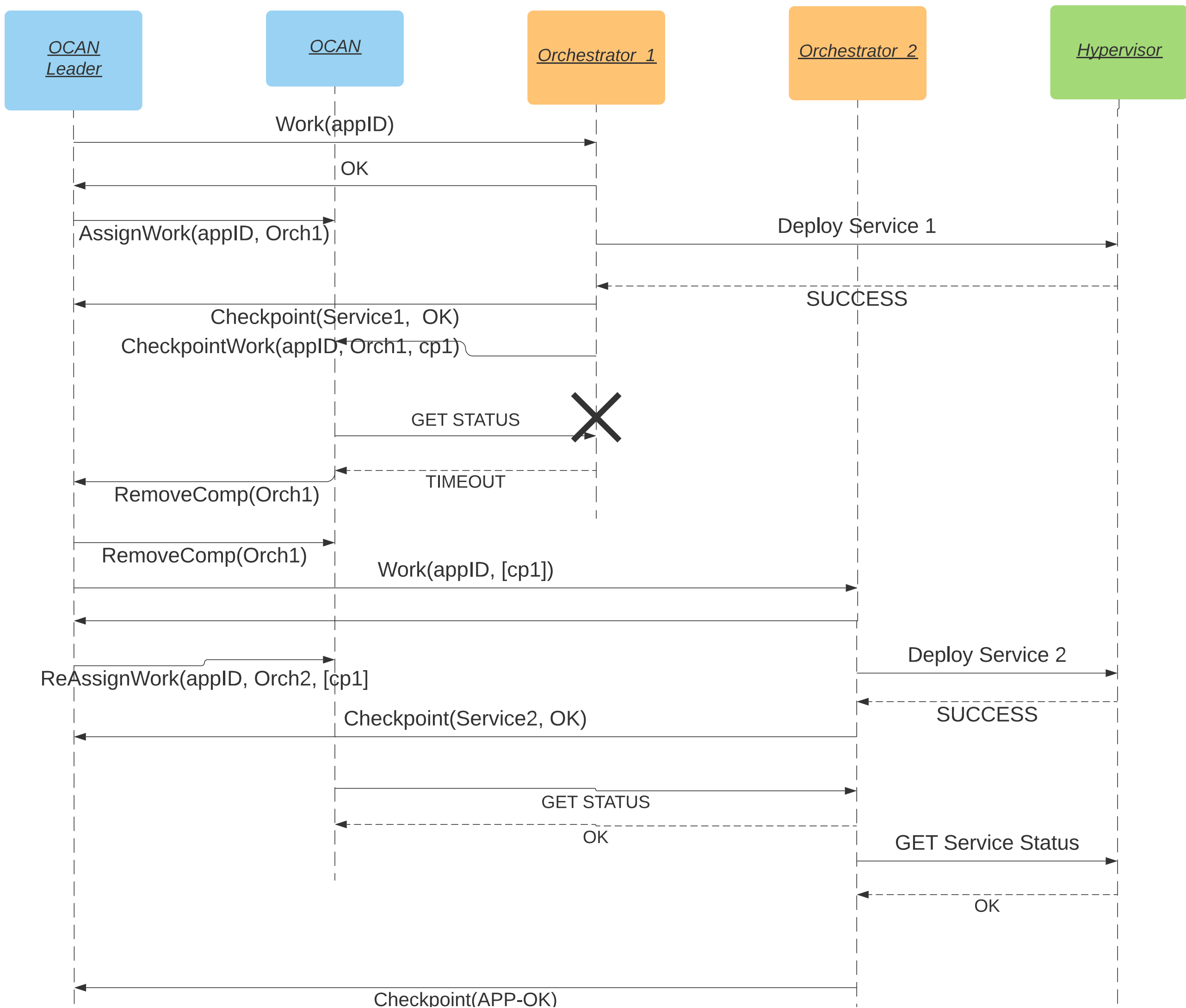


Fig. 5.1. *Example deployment continuity with failing Orchestrator*

The Algorithm starts from the first non-paid checkpoint, and for each checkpoint, the type is inspected. A *SERVICE_UP* checkpoint just sets the start time of a Service. A *SERVICE_DOWN* checkpoint leads to computing the cost for the corresponding Service, and if sufficient funds are available the *salary* map is updated using the *payService* function (described in Algorithm 6). An *APP_OK* checkpoint means all Services are executing and prepares payment for all Services of the Application. If there are not sufficient funds to pay for all the Services, then this checkpoint can not be paid and the loop breaks. In the successful case, the start time of service is updated to be the timestamp of this checkpoint; a further *SERVICE_DOWN* or *APP_OK* checkpoint will be paid in relation to the current one, as the rest has already been paid. Finally, *lp* is updated to the new last paid checkpoint index and the total payable funds are added to a variable, *locked* which restricts the client to withdraw funds that have been promised to be paid. In the end, the total number of paid checkpoints is returned to the OCAN leader. The *APP_SHUTDOWN* checkpoint initiates the same behaviour as the *APP_OK* checkpoint, with the exception that the start time for each service is set to *null*.

The *payService* function computes the reimbursement for each entity and updates the salary map. The Cloud, Physical Resources, and Orchestrators are paid individually, while the OCAN reimbursed trough the

Algorithm 5 Interim Payment Function

```
1: cp – checkpoint array
2: lp – last paid checkpoint
3: salary – map < address, int > with payments
4: start – map < string, int > map with start times per service
5: function interimPayment(k)
6:     newlp ← lp
7:     total ← 0
8:     for i ← lp + 1 to min(lp + k, cp.size) do
9:         c ← cp[i]
10:        if c.type = SERVICE_UP then
11:            start[c.s_name] = c
12:        else
13:            if c.type = SERVICE_DOWN then
14:                x ← computePayment(start[c.s_name], c)
15:                if x + total < getBalance() then
16:                    payService(start[c.s_name], c)
17:                    total+ = x; newlp+ = 1
18:                    start[c.s_name] ← null
19:                else
20:                    break
21:                end if
22:            else
23:                if c.type = APP_OK then
24:                    fail ← False; appTotal ← 0
25:                    for s in services do
26:                        appTotal+ = computePayment(start[s.name], c)
27:                        if total + appTotal > getBalance() then
28:                            fail ← True
29:                            break
30:                        end if
31:                    end for
32:                    if fail then break
33:                    end if
34:                    for s in services do
35:                        payService(start[s.name], c)
36:                        start[s.name] ← c
37:                    end for
38:                    newlp+ = 1
39:                    total+ = appTotal
40:                end if
41:            end if
42:        end if
43:    end for
44:    steps ← newlp − lp; lp ← newlp; locked+ = total
45:    return steps
46: end function
```

Registry contract, which maintains information about the historical availability of each node in the OFTEN. Different Orchestrators can be the contributors of to two consecutive checkpoints; in this case, each of them

gets paid half the orchestration price. In our example, we have considered a per-minute based pricing, and timestamps logged in milliseconds.

Algorithm 6 Service Payment Function

```
1: function PAYSERVICE(cs, ce)
2:     time ← (ce.timestamp − cs.timestamp)/1000/60
3:     res ← Cloud.resourcePrice · time
4:     cloud ← Cloud.price · time
5:     ften ← Registry.ftenPrice · time
6:     orc ← orcPrice · time
7:     salary[Cloud.address]+ = cloud
8:     salary[services[ce.s_name].resAddress]+ = res
9:     salary[RegistryAddress]+ = ften
10:    if ce.orcAddress ≠ cs.orcAddress then
11:        salary[cs.orcAddress]+ = orc/2
12:        salary[ce.orcAddress]+ = orc/2
13:    else
14:        salary[ce.orcAddress]+ = orc
15:    end if
16: end function
```

The *APP_SHUTDOWN* checkpoint can also be issued in the case several interim payments have been tried with 0 successful checkpoints paid. The shutdown checkpoint will trigger the release of the associated resources, marking them as available in the corresponding Cloud Contract. If an Application is in a SHUTDOWN phase, the Application Contract can be destroyed to release space on the Ethereum Blockchain. If salaries have not been collected, the client will support the cost of sending the salaries to the accounts of the entitled entities when requesting the destruction of the Contract. Any unused funds are returned to the client.

6. Related work. Several companies are tackling the offering of Cloud Services through the means of Blockchains, with limited scientific output.

Golem[2] makes use of IPFS [8] to distribute input file blocks in the worker nodes network. Workers are processing data at the block level and all the parallel results will be merged at the user's machine. Compared with them, our proposal is more generic, allowing for user-defined Applications.

SONM[3] achieves a higher level of abstraction, using Docker for executing Container Images. An Ethereum-based side chain is used for managing *Orders*. *Suppliers* interact with the side chain to set up workers acting on their behalf. The workers expose resources such as CPU, RAM, storage, bandwidth in the form of *benchmark identifies*, e.g., 20 *GFLOPS*. A user can rent some resources for a limited amount of time, or on a pay-as-you-go model. There is a limited amount of documentation concerning how the system is matchmaking user requests with resources. In comparison with SONM, our Orchestration process allows for the deployment of Applications as Virtual Machines, Docker Container or Bare Metal infrastructure. Additionally, it supports hardware accelerators (GPUs, MICs) that improve the performance of the Applications. Moreover, this paper proposes a fault tolerance enforcing mechanism for the management Components. SONM does not make clear what mitigation actions are in place for dealing with node failures.

The **iExec**[4] platform is based on renowned of research in volunteer computing [12, 29]. The Ethereum Blockchain is used to manage the platform tokens, and a side chain is used and implement the platform logic. When an application is requested, multiple nodes will execute it and the results are compared. A *Proof of Contribution* (*PoCo*) protocol is used for acknowledging the correct result of an Application, using the sabotage tolerance introduced in [13]. The *PoCo* links two entities: the iExec marketplace (where deals are made) and

[2] https://golem.network/
[3] https://docs.sonm.com/concepts/main-entities
[4] https://iex.ec

the computing infrastructure (based on XtremWeb-HEP middleware [12]). Compared to iExec, our proposal is again more generic. iExec Applications must have a final results, which executing nodes should agree on, while in our proposal each Service should provide a status endpoint which the Orchestrator can check to see if the Service is available.

7. Conclusion. This paper proposed an architecture and mechanism for decentralized Orchestration of Cloud Service on resources residing in homes or small-scale clusters. The proposed framework is able to enforce the fault tolerance of the Orchestration process and to assess the execution time for a Service. The CloudLightning architecture has been conceived to provide efficient and flexible deployment of HPC-aware Services, managing the infrastructure within a data centre. This architecture is augmented to decentralize the Resource Management and Service Orchestration processes, which are synchronized through Ethereum Smart Contracts.

Component Administration Networks provide a bridge from the Smart Contract world to the software world and ensure the fault tolerance of supervised components. Such a network stores checkpoints for the supervised components which allow a new replica to continue the work if another had crashed. Simulation results show that failure rates of up to 75% can be tolerated among the CAN nodes if enough candidate nodes are willing to join the network periodically. This is encouraged by reimbursing the nodes that are part of this network. We have demonstrated the usage of an Orchestration Component Administration Network to ensure the continuity of Application deployment. Additionally, in conjunction with the Application Contract, the checkpoints are used to assess the amount of time different entities in the system have dedicated for the execution of the Application and ensure a fair price for the user and a fair reimbursement all participants.

The proposed architecture and mechanisms pave the way for a decentralized free-market, where individually owned resources meet the demands for Cloud Applications. In the realm of multiple Cloud Contracts, resources prefer the Cloud with the highest pay, but clients prefer Clouds with the lowest price. Resources are free to move to Clouds that may pay less, but more often, and some users may pay a higher price for more efficient hardware.

REFERENCES

[1] Urs Hölzle and Luiz André Barroso. Warehouse-scale computers. *IEEE Internet Computing Magazine*, 14(1):33, 2010.

[2] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[3] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[4] Bernadette Charron-Bost, Fernando Pedone, and André Schiper. Replication. *LNCS*, 5959:19–40, 2010.

[5] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[6] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[7] Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys (CSUR) **22**(4), 299–319 (1990)

[8] Benet, J.: Ipfs-content addressed, versioned, p2p file system. arXiv preprint arXiv:1407.3561 (2014)

[9] Maymounkov, P., Mazieres, D.: Kademlia: A peer-to-peer information system based on the xor metric. In: International Workshop on Peer-to-Peer Systems, pp. 53–65. Springer (2002)

[10] Pouwelse, J., Garbacki, P., Epema, D., Sips, H.: The bittorrent p2p file-sharing system: Measurements and analysis. In: International Workshop on Peer-to-Peer Systems, pp. 205–216. Springer (2005)

[11] Anderson, D.P.: Boinc: A system for public-resource computing and storage. In: proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, pp. 4–10. IEEE Computer Society (2004)

[12] Fedak, G., Germain, C., Neri, V., Cappello, F.: Xtremweb: A generic global computing system. In: Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on, pp. 582–587. IEEE (2001)

[13] Sarmenta, L.F.: Sabotage-tolerance mechanisms for volunteer computing systems. Future Generation Computer Systems **18**(4), 561–572 (2002)

[14] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. URL https://bitcoin.com/bitcoin.pdf (2008)

[15] Buterin, V., et al.: Ethereum white paper, 2014. URL https://github. com/ethereum/wiki/wiki/White-Paper (2013)

[16] Protocol Labs: Filecoin: A Decentralized Storage Network URL https://filecoin.io/filecoin.pdf [Accessed: 8-12-2020].

[17] Fisch, B., Bonneau, J., Greco, N., Benet, J.: Scaling proof-of-replication for filecoin mining. Tech. rep., Technical report, Stanford University, 2018. https://web. stanford. edu … (2018)

[18] Talal Ashraf Butt, Razi Iqbal, Khaled Salah, Moayad Aloqaily, and Yaser Jararweh. Privacy management in social internet of vehicles: Review, challenges and blockchain based solutions. *IEEE Access*, 7:79694–79713, 2019.

[19] Geetanjali Rathee, Ashutosh Sharma, Razi Iqbal, Moayad Aloqaily, Naveen Jaglan, and Rajiv Kumar. A blockchain framework for securing connected and autonomous vehicles. *Sensors*, 19(14):3165, 2019.

[20] Yehia Kotb, Ismaeel Al Ridhawi, Moayad Aloqaily, Thar Baker, Yaser Jararweh, and Hissam Tawfik. Cloud-based multi-agent cooperation for iot devices using workflow-nets. *Journal of Grid Computing*, pages 1–26, 2019.

[21] Moayad Aloqaily, Ismaeel Al Ridhawi, Haythem Bany Salameh, and Yaser Jararweh. Data and service management in densely crowded environments: Challenges, opportunities, and recent developments. *IEEE Communications Magazine*, 57(4):81–87, 2019.

[22] T. Lynn, H. Xiong, D. Dong, B. Momani, G. Gravvanis, C. Filelis-Papadopoulos, A. Elster, M. Khan, D. Tzovaras, K. Giannoutakis, D. Petcu, M. Neagul, I. Dragon, P. Kuppudayar, S. Natarajan, M. McGrath, G. Gaydadjiev, T. Becker, A. Gourinovitch, D. Kenny, and J. Morrison. Cloudlightning: A framework for a self-organising and self-managing heterogeneous cloud. In *the 6th International Conference on Cloud Computing and Services Science*, volume 1, pages 333 - 338, 2016.

[23] Christos Filelis-Papadopoulos, Huanhuan Xiong, Adrian Spataru, Gabriel G Castañé, Dapeng Dong, George A Gravvanis, and John P Morrison. A generic framework supporting self-organisation and self-management in hierarchical systems. In *Parallel and Distributed Computing (ISPDC), 2017 16th International Symposium on*, pages 149–156. IEEE, 2017.

[24] Christos K Filelis-Papadopoulos, Konstantinos M Giannoutakis, George A Gravvanis, and Dimitrios Tzovaras. Large-scale simulation of a self-organizing self-management cloud computing framework. *The Journal of Supercomputing*, 74(2):530–550, 2018.

[25] Paul Stack, Huanhuan Xiong, Dali Mersel, Maxime Makhloufi, Guillaume Terpend, and Dapeng Dong. Self-healing in a decentralised cloud management system. In *Proceedings of the 1st International Workshop on Next generation of Cloud Architectures*, pages 1–6, 2017.

[26] Adrian Spataru, Laura Ricci, Dana Petcu, and Barbara Guidi. Decentralized cloud scheduling via smart contracts. operational constraints and costs. In *The International Symposium on Blockchain Computing and Applications (BCCA2019)*, 2019.

[27] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[28] CloudLightning. CloudLightning deliverable D5.3: GATEWAY SERVICE . http://cloudlightning.eu/work-packages/public-deliverables/, 2017. Accessed: 2020-11-04.

[29] Mircea Moca, Cristian Litan, Gheorghe Cosmin Silaghi, and Gilles Fedak. Multi-criteria and satisfaction oriented scheduling for hybrid distributed computing infrastructures. *Future Generation Computer Systems*, 55:428–443, 2016.

Edited by: Viorel Negru
Received: Nov 19, 2020
Accepted: Dec 18, 2020