Article

A Blockchain Protocol for real-time Application Migration on the Edge

Aleksandar Tošić [1,†,‡] [3,*], Jernej Vičič [1,‡] [2,], Michael Burnard [4,5], Michael Mrissa [6,7]

[1] University of Primorska Faculty of Mathematics, Natural Sciences and Information Technologies; aleksandar.tosic@upr.si
[2] University of Primorska Faculty of Mathematics, Natural Sciences and Information Technologies; jernej.vicic@upr.si
[3] InnoRenew CoE; Livade 6 6310 Izola, Slovenia; aleksandar.tosic@innorenew.eu
[4] InnoRenew CoE; Livade 6 6310 Izola, Slovenia; mike.burnard@innorenew.eu
[5] Institute Andrej Marušič; Muzejski trg 2, 6000 Koper, Slovenia; mike.burnard@iam.upr.si
[6] InnoRenew CoE; Livade 6 6310 Izola, Slovenia; michael.mrissa@innorenew.eu
[7] University of Primorska Faculty of Mathematics, Natural Sciences and Information Technologies; michael.mrissa@upr.si
* Correspondence: aleksandar.tosic@upr.si;
† Current address: Glagoljaška 8, 6000 Koper, Slovenia
‡ These authors contributed equally to this work.

Abstract: The Internet of Things (IoT) is experiencing widespread adoption across industry sectors ranging from supply chain management to smart cities, buildings, and health monitoring. However, most software architectures for IoT deployment rely on centralized cloud computing infrastructures to provide storage and computing power, as cloud providers have high economic incentives to organize their infrastructure into clusters. Despite these incentives, there has been a recent shift from centralized to decentralized architecture that harnesses the potential of edge devices, reduces network latency, and lowers infrastructure cost to support IoT applications. This shift has resulted in new edge computing architectures, but many still rely on centralized solutions for managing applications. A truly decentralized approach would offer interesting properties required for IoT use cases. In this paper, we introduce a decentralized architecture tailored for large scale deployments of peer-to-peer IoT sensor networks and capable of run-time application migration. We propose a leader election consensus protocol for permissioned distributed networks that only requires one series of messages in order to commit to a change. The solution combines a blockchain consensus protocol using Verifiable Delay Functions (VDF) used for decentralized randomness, fault tolerance, transparency, and no single point of failure. We validate our solution by testing, and analyzing the performance of our reference implementation. Our results show that nodes are able to reach consensus consistently, and the Verifiable Delay Function proofs can be used as an entropy pool for decentralized randomness. We show our system can perform autonomous real-time application migrations. Finally, we conclude that the implementation is scalable by testing it on 100 consensus nodes running 200 applications.

Keywords: Fault Tolerance; Blockchain; Internet of Things; Edge Computing; Peer-to-Peer; Decentralized; Sensor Networks; Verifiable Delay Functions

---

1. Introduction

Cloud computing solutions have driven the centralization of computing, process control (e.g., business information, manufacturing, distributed systems, IoT management), and data storage to data centres. Existing cloud-based solutions have few incentives, aside from reducing network latency, to distribute computing and storage resources. There are many reasons why centralization is more appealing. These range from legislative reasons, tax policies, availability and affordability of high speed internet and electrical power, reduction of maintenance costs, and even climate preservation [1].

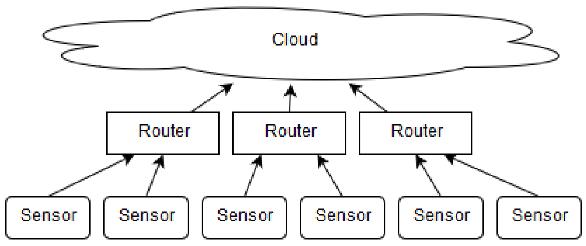

Figure 1. Standard sensor network architecture.

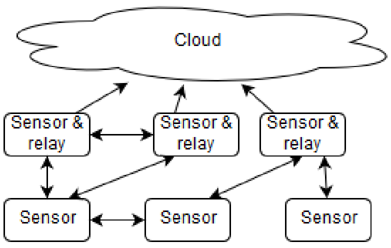

Figure 2. Mesh sensor network architecture.

However, cloud computing solutions are struggling to address the specific challenges of emerging IoT, and edge computing use cases.

The ever-growing number of devices on the edge causes scalability challenges for centralized architectures such as cloud-based ones. Edge devices tend to be heterogeneous and existing IoT platforms remain isolated and unable fully exploit their potential. Moreover, these devices have considerable computing resources, which for the most part remain underutilized as most applications perform computation on the cloud. A major challenge in this area is supporting homogeneous usage of edge devices, which requires applications to migrate at run-time from an overloaded device to a more available one. Currently, there is no standardized platform for general purpose computing that supports such run-time application migration. Another limitation to large scale deployment of sensor networks is the infrastructural investment needed to support the network, as typical architectures require a middle layer infrastructure that enables access to the cloud (Fig. 1 and [2]).

We believe these challenges can be overcome as recent technological advances have provided partial solutions and have presented new opportunities. These advances have paved the way for the recent paradigm shift from centralized to decentralized architectures for IoT [3]. First, as edge devices are becoming more powerful and capable of running complex software, they provide a huge pool of available, yet underutilized, computing resources. Second, containerization solutions (as opposed to virtualization) have been gaining momentum to overcome heterogeneity problems while preserving acceptable performance. Containerization software (e.g., Docker) provides software abstraction that enables general purpose computing on edge devices. Third, with the growth of edge devices capable of direct wireless communication, a mesh network approach has become worth exploring as a solution to reduce or eliminate the middle layer infrastructure needed for devices to connect to each other and the cloud (Fig. 2).

The need for edge computing is well illustrated with scenarios related to ad-hoc networks [4], and especially with peer-to-peer wireless sensor networks. The paradigm shift towards decentralization is relevant to numerous application domains such as smart building monitoring, structural health monitoring, self-driving vehicles, micro service architectures, mobile devices, etc.. In our work, we have experimented with a cultural heritage building located in Bled, Slovenia. We deployed several sensors to monitor the building state for maintenance purposes and air quality to provide safety for visitors. In the case where buildings are located in remote areas, as in our use case, edge devices must self-regulate and optimize their behaviour at run-time. They also must have the capacity to scale up as the number of devices grows (scalability), to adjust when dysfunctions occur, for example when devices leave the network (experience byzantine behaviour), and the operation of all devices should be recorded safely for later analysis (transparency). In a cloud-based environment, edge devices send data to the cloud where computation occurs. However, issues such as poor network coverage, frequent disconnection, cost of infrastructural investment, inadequate dependability, and security concerns remain unaddressed [5], [6], and [7]. Edge computing solutions attempt to reduce network latency, increase fault tolerance, dependability and security, and reduce the cost of infrastructural investment needed to provide network coverage. They also operate independently of an external network connection.

To address these issues, we propose an architecture based on an innovative combination of existing technologies. Specifically, our architecture provides a general purpose computation model allowing large scale sensor networks to distribute the computational load among edge devices (sensors, controllers, etc.). Using containerization, applications can be built using any programming language or stack, containerization also serves as an abstraction layer between the application requirements, and the hosts hardware. The decision making process for resource allocation is made by a decentralized orchestrator implemented as a consensus protocol that outputs a migration strategy, which is in turn stored on the blockchain [1]. It features high fault tolerance, full transparency, reduced network infrastructure cost, and no single point of failure. The network layer uses decentralized randomness to constantly change the network topology to allow efficient propagation of information pertaining to resource utilization of nodes.

The rest of this paper is structured as follows: Section 2 provides the necessary background knowledge and overviews the most relevant related works to highlight the originality of our proposal. Section 3 details our architecture and its operation. Section 4 describes our evaluation environment Section 5 summarizes the results and gives guidelines for future work.

2. Background knowledge and related work

The most critical unmet monitoring challenges according to [8] are: mobility management, scalability and resource availability at the edge of the network, coordinated decentralization, interoperability and avoiding vendor lock-in, optimal resource scheduling among edge nodes, and fault tolerance. No widely-used cloud monitoring tool for edge computing fully addresses these challenges. some requirement remain unmet by any existing solution, as many system aspects including container, end-to-end network quality are not adequately addressed [8]. The EU project RECAP [9] presents a vision of the next generation of intelligent, self-managed, and self-remediated cloud computing systems (i.e., a system that can monitor and relocate resources to achieve Quality of Service - QoS). The project also describes models intended to be integrated in network topology-aware application orchestration and resource management systems from an edge computing perspective [10]. Another solution, AutoMigrate [11], incorporates a selection algorithm to determine which services should be migrated to optimize availability. Although this system has solutions for most of the problems we address, it does not resolve the Single Point Of Failure (SPOF) issue because it relies on a central service to orchestrate migrations. Our decentralized implementation eliminates the SPOF issue.

2.1. Orchestration solutions for edge computing

By definition, orchestration denotes control by a single entity over many. This differs from choreography, which is more collaborative and allows each involved party to describe its part in the interaction [12]. We have identified the most successful orchestration solution to be Kubernetes [13],

[1] A blockchain is a growing list of records called blocks, linked together using cryptography, and the nodes follow a shared consensus protocol to validate new blocks.

the most used and most feature-rich orchestration tool [14], Docker Swarm[2], Amazon Web Service Elastic Container Service (AWS ECS) [15], the Distributed Cloud Operating System[3], and Nomad[4].

The Decenter EU project[5] proposes decentralized orchestration technologies for fog-to-edge computing. Although the project does support decentralized orchestration between multiple domains and records service level agreements and violations to the blockchain, the solution is designed as a federated approach where a multi-domain orchestrator overviews several domains, that in turn are driven from local orchestrators [16]. The project also implements a blockchain to act as a brokerage platform where smart contracts guarantee resource sharing across domains [17]. In contrast to a federated approach, our implementation is fully decentralized with a randomly selected orchestrator at each interval, thus avoiding the SPOF problem and not relying on a trusted third party.

All of the architectures discussed above have a common flaw: the SPOF problem. In each case, the flaw is characterised by a single orchestration entity. Most solutions also lack support for edge devices. Our proposed solution addresses these shortcomings, while providing full transparency, variability of the system, completely decentralized operation backed with a strongly secure, scalable, and efficient consensus mechanism.

Recently, a decentralized protocol for orchestration of containers named Caravela was proposed [18]. The solution relies on a Chord for resource discovery, and employs a volunteer system in which nodes are categorized as suppliers (supplying resources), buyers(searching for resources), and traders(mediating supply/search for offers). The authors show that their solution can scale using a random migration algorithm, but fails fulfill deployment requests. It also is not able to fulfill the global binpack scheduling policy due to a lack of global shared state.

2.2. Container platforms

We are using containers as a primary execution environment. Containers, as used in this paper, are a group of namespaced processes run within an operating system. Docker is the most widely used platform according to [19] and one of the few platforms that can migrate apps at run-time and enable easy communication. For these reasons it was used as the main testing platform.

2.3. Available blockchain solutions

The proposed solution makes use of a blockchain to store the state transitions of the network in a verifiable, and transparent way. Unlike existing blockchains which either use an account based model [20], or an UTXO model [21], our blocks do not store transactions or account states. The block structure is tailored to accommodate application migration and verifiability of migrations. Hence, the blocks are snapshots of the state of the system containing information about available resource, and required resources of applications managed by the system.

A survey of the most notable readily available blockchain solutions for private network yielded three candidates:

- Implementation of a private Ethereum network, although the implementation is fairly simple [22], the available consensus mechanisms include PoW, which is not secure for networks with no value, and proof of authority (PoA), which limits the consensus nodes to a subset of trusted nodes thereby decreasing decentralization, and security.
- Implementation of a HyperLedger blockchain in all configurations requires notable CPU burdens [23]. As the number of nodes in the network grows, the system requirements scale far beyond what can be considered sustainable for edge devices.

[2] https://github.com/docker/swarm
[3] https://dcos.io/
[4] https://www.hashicorp.com/products/nomad
[5] Decenter project homepage: https://www.decenter-project.eu

- Multichain [6] also presents a viable alternative for a private blockchain network [24], again not suitable for Edge devices [24]. Moreover, it is primarily focused on facilitating transactions of cryptocurrency, and assets.
- Solana [25] similarly uses verifiable delay functions as a source of entropy for their leader rotation algorithm. However, their VDF implementation requires thousands of graphical processing units to meet the speed requirements, which is not suitable for edge devices.

All the presented available off-the-shelf solutions satisfy most of the criteria posed by the research experiment, but they all rely heavily on the computation power which makes them unsuitable for edge devices. Further, the required block structure and changes on the protocol would outweigh the benefits and accumulate technical debt.

2.4. Decentralized self-managing IoT architectures

A survey of the scientific literature shows multiple solutions that address decentralized self-managing architectures for the IoT. The most notable examples are:

- Maior et al. [26] present a theoretical description of a decentralized solution for energy management in IoT architectures. The solution is aimed at smart power grids. They present 4 algorithms with analyses of correctness in order to describe the behavior of self-governing objects.
- Higgins et al. [27] propose a distributed IoT approach for electrical power demand management.
- Suzdalenko and Galkin [28] extend the approach by Higgins et al. [27] by allowing users to individually join, and depart the environment at run-time.
- Niyato et al. [29] propose a system that addresses home energy management system wheres devices communicate directly among themselves.
- dSUMO [30] address the synchronization bottleneck by proposing a distributed and decentralized microscopic simulation (the focus is on data throughput and not fault tolerance; throughput is increased using a decentralised setting).
- Al-Madani et al. [31] address indoor localization utilizing Wireless Sensor Networks (WSNs) relaying on publish/subscribe messaging model. The results show that the Really Simple Syndication (RSS) [32] format achieves acceptable accuracy for multiple types of applications.

Our proposed solution differs from the previous contributions in two ways.

- other solutions typically focus on a single problem presenting an optimal solution for it, we argue that an IoT architecture requires multiple optimisation criteria. We consider multiple criteria and include a framework to add more criteria in the future.
- our protocol is highly decentralized as it allows all nodes to participate in the consensus, while maintaining low hardware requirements fit for edge devices

A related approach by Samaniego and Deters [33] suggests using virtual resources in combination with a permission-based blockchain for provisioning IoT services on edge hosts. They use blockchain to manage permissions only, and therefore provide security using blockchain. In contrast, our approach uses blockchain to store all information about service choreography which makes it verifiable over time, while still providing security.

The main contribution of this paper is a light-weight blockchain protocols, which can achieve high decentralization and low hardware requirements typically found in edge devices. The proposed protocol inherits ideas from Ethereum 2.0 but replacing the source of entropy needed for consensus with a VDF function. Moreover, the structure of the block carries the state transition information, and unlike existing blockchains does not have the concept of account, balances, and transactions.

[6] MultiChain Open source blockchain platform: https://www.multichain.com/

3. Proposed decentralised architecture

In this section, we provide a general description of our architecture [34] and highlight its main components. The main purpose of our architecture is to enable verifiable and decentralized management of applications on the edge. In our vision, applications can be built as containers, and submitted to the network by reaching any node via an API. We use containerization to decouple the host running the application from the application and address the issue of hardware and software heterogeneity. This allows the protocol to assume an application can be run on all nodes in the network. A randomly selected and decentralized orchestrator on the network would then be able to choreograph the execution of the application, and migrate applications between hosts at run-time. As our architecture is fully decentralized, each node is locally driven by a protocol that participates in establishing the global state of the network via a specially built consensus mechanism. Nodes in the network reach consensus on a migration plan in an effort to improve the resource allocation of running applications. A migration plan is viewed as a state transition, which is stored on the blockchain formed by the participating nodes. We implement a choreographed solution, which is a collaborative, rather than a directed approach (as opposed to orchestration). Choreographed systems define a way for each member to describe its role in the interaction [12]. This collaborative approach avoids the SPOF problem. Despite this advantage, there are no choreography solutions known to address the open problems described in the start of the Section 2.

To provide a global understanding of our fully decentralized architecture, we first describe the architecture of a single node, followed by the interaction protocols between nodes.

All identified orchestration solutions presented in Section 2.1 rely on a primary/replica model. The main service selects the applications that need to be reallocated according to a selected optimization algorithm.

Our migration algorithm is able to:

- pause a container,
- transfer the context to a different host,
- resume the execution given the context.

Additionally, we implemented migrations using checkpoint/restore in userspace, or CRIU, an experimental feature available in Docker [35].

3.1. Overview of node architecture

The node application that we developed is containerized in Docker. As shown in Fig. 3, the internal architecture of a node is composed of the following modules that support application management:

- Networking layer: this layer deals with network communication through deployed APIs.
- Gossip protocol: A rendezvous based gossip protocol is used to build a distributed hash table (DHT) that maps public IP's of nodes to their network address. The messages are encoded using protocol buffers[7], it is the underlying protocol that makes sure all messages reach all nodes in the network while minimising network usage.
- Block propagation protocol: relies on the gossip protocol to spread newly accepted blocks over the network.
- Resource propagation protocol: relies on the network layer to deliver the state of resources (currently CPU, RAM, disk usage, and network utilization of Docker containers) over the network to the receiving node.
- Consensus protocol: ensures all nodes reach consensus in a decentralized way (presented below).

[7] https://developers.google.com/protocol-buffers

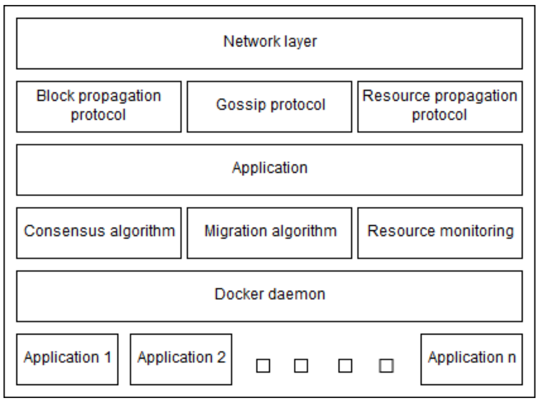

Figure 3. General overview of a node architecture.

- Migration algorithm: guarantees that a migration strategy is reached whenever needed thanks to a deterministic algorithm. This algorithm is executed at each slot until the proposed block is accepted and finalized. The output of the algorithm is included in the block to construct a verifiable, and transparent log of each application's life-cycle.
- Docker daemon: hosts applications and is used for abstracting the underlying heterogeneity between devices, systems, and applications. It provides support to our solution via Docker APIs.
- Resource monitoring: relies on the Docker API to monitor the state, and resource allocations of the hosting device and the applications running on it.

3.2. Storing system states in a blockchain

All nodes share information about their states through a federated type topology obtained by distributed clustering of nodes explained in more detail in Section 3.3. We define a state as a matrix of vectors describing resource consumption associated with each application. A *resource pool* data structure is replicated in all nodes and contains information about all node states. In our use case, we define a vector with the following values:

$$\{app, cpu, ram, disk, network, timestamp\}$$

This provides a time series of system resource utilisation for each application across an operating period. The resources used by applications are obtained through the Docker API and represented in percentages for simplicity. Taken at a specific time interval, the vector is a block that includes a list of per-application resource statistics, as shown in Table 1 where *Node* is a 256bit hash representing the system wide unique ID of the application, RAM, DISK, and CPU are floats representing the portion of node's available resources used by the application. Finally, the average latency is computed as the 30 second moving average of Round-trip delay (RTT) towards randomly selected validators.

Table 1. An example of a data block.

V	Node	RAM	DISK	CPU	Average Latency
v_0	A	50%	23%	90%	23ms
v_1	B	47%	87%	23%	33ms
v_2	C	12%	25%	15%	51ms
v_3	A	35%	14%	56%	101ms
v_4	D	25%	74%	16%	9ms

From a data block, it is then possible to compute a migration plan to optimize the allocation of applications to nodes according to the resource states of all nodes. The migration plan is also included in the block, which produces a transparent computational log for verifying if the adopted migration plan was actually efficient and fair. The architecture does not enforce any specific migration algorithm. The only constraints are that the algorithm must be deterministic and must rely only on data included in the block (reached by consensus). For the same inputs to a deterministic algorithm, proposed migrations can be verified much like transactions are verified in public blockchains.

In order to provide liveliness and responsiveness, delivering resource consumption statistics to the block producer must be faster then $\frac{2}{3} * slotTime$. Using the gossip protocol produces unwanted latency, and greatly increases resource utilization maintaining the message queue (MQ). To overcome this, we implement a distributed k-means clustering algorithm that requires no communication between nodes to compute. Clustering is used to group nodes in a separate overlay network where statistics are propagated using UDP protocol. The seed used to compute k-means is shared by all nodes, the VDF proof. Cluster representatives are nodes that are responsible for requesting resource utilization statistics from their members, and transmitting them to the block producer. The timing details are strongly intertwined with the synchronicity of the consensus algorithm further explained in chapter 3.4.

This federated overlay topology greatly decreases decentralization and consequently fault tolerance in case a cluster representative exhibits byzantine behaviour. However, this is not concerning considering that a failure to disseminate resource utilization only delays potential migrations for a subset of applications in the current block. Once a new block is accepted, a new overlay topology is computed. Eventually a node experiencing byzantine behaviour is excluded from the validator set as detailed in chapter 3.4.

3.3. Migration algorithm and verifiability

To forge a block, nodes compute a migration plan based on resource statistics in the previous block. The migration plan is executed once the proposed block is accepted. Application migration is realized using Docker commands to pause the application, compress it, and transfer it to the destination node where it is restored. Alternatively, using CRIU, only the state of the running container is extracted, and migrated. All migration plans are securely stored in the blockchain for eventual verification. The time to produce a block is configurable, and largely depends on the requirements for responsiveness, and resource availability, and network size. However, there are some lower bounds set by the consensus protocol (empirically, 5 seconds), under which we experience occasional block propagation issues, vote propagation, and aggregation delays that can cause unplanned soft forks.

Each block contains data that describes the states of nodes and the migration plan resulting from the application of the generation algorithm. It also contains the signature of the previous block to follow the principles of the blockchain, so that all blocks are dependent on the previous blocks, which makes it irreversible. To demonstrate our approach, we relied on the sample algorithm presented in Table 1 to generate migration plans according to the resource pool. Blocks also include meta-data that facilitate their utilization such as block hash, previous block hash, VDF proof, aggregated votes, validator set updates, slot, and epoch.

3.4. Consensus mechanism

A key component of a blockchain is the ability for nodes to reach consensus on the global state of the ledger. With increasing interest in blockchain technology in recent years, many consensus algorithms have built upon basic proof of work[8] concepts. However, most algorithms used in permissionless blockchain implementations rely on basic game theory assumptions, which hold only when the blockchain facilitates value transfers, where we can rely on actors acting according to their

[8] Proof of work [36] is a technique that protects from various attacks by requiring a certain amount of processing power to use a service, which makes a potential attack worthless because it becomes too costly.

Algorithm 1 Deterministic migration plan generation.

```
Input: BlockData
Output: Generation plan
  $Max \Leftarrow FindMaxLoadedNode(BlockData)$
  $Min \Leftarrow FindMinLoadedNode(BlockData)$
  if !AppQueue.isEmpty() then
    while !AppQueue.isEmpty() do
      $Min \Leftarrow FindMinLoadedNode(BlockData)$
      $Min.addApp(AppQueue.dequeue())$
    end while
  else
    $AppToMigrate \Leftarrow Max.MaxLoadApp$
    $DeltaScore \Leftarrow (Max.score - Min.score)$
    $NextDeltaScore \Leftarrow (Max.score - AppToMigrate.score) - (Min.score + AppToMigrate.score)$
  end if
  if $Math.abs(DeltaScore > NextDeltaScore)$ then
    Migrate($AppToMigrate, Min$)
  end if
```

own (financial) interests (i.e., the nothing at stake problem). In permissioned networks, where there is usually no monetary value, the consensus algorithms used in monetary blockchain implementations are not appropriate. Instead, a known family of consensus algorithms for permissioned networks can be used based on voting schemes for leader elections like PBFT [37], Proof of Elapsed Time (PoET) [38] or RAFT [39]. However, these algorithms require multiple messages to be sent through the network in order to commit a change.

Our algorithm is based on a random draw that is universally verifiable. To achieve decentralized randomness and verifiability, we make use of Verifiable Delay Functions (VDF) [40]. A VDF is a function that takes a large quantity of non-parallel work to compute, and produces a verifiable proof. More specifically, VDFs are similar to time lock puzzles but require a trusted setup where the verifier prepares each puzzle using its private key. Additionally, a difficulty parameter can be adjusted to increase the amount of sequential work, thereby increasing the delay. We extend our previous consensus algorithm [34] such that nodes first compute a VDF depending on the difficulty assigned for block $n+1$, and desired *slotTime*, which is a configurable parameter of the network. We then use the proof $P_n = VDF((n-1)_{hash}, (n-1)_{difficulty})$ as a decentralized entropy pool for random number generator(RNG) to draw decentralized randomness for a given *slot*. For every slot, nodes are able to self-elect into consensus roles (e.g., *Block Producer*, *Validator*, *Committee Member*) as outlined in Algorithm 2. Due to the seeded RNG, all nodes compute the same assignment of roles for all participating nodes thereby not requiring any message exchange to agree on their roles. Moreover, the canonical nature of the chain provides some security guarantees so that the roles for future block $n+1$ cannot be computed before block n is accepted. Once roles are assigned for a given slot, nodes perform their sub-protocols as follows:

1. *Block Producer* is a singular node elected each slot to produce a candidate block. The candidate block is sent to all committee members. Upon sending, the block producer listens for attestations for $\frac{2}{3} * slotTime$, and aggregates them. The aggregated signature is then included in the block header, and gossiped to the entire network if a sufficient number of votes are received, otherwise a skip block is proposed.
2. *Committee Member* are responsible for attesting to candidate blocks. They verify the block integrity, signatures, and data to produce a Boneh-Lynn-Shacham signature (BLS), then send the signature to the block producer.
3. *Validator* nodes receive a new block, verify the integrity and committee signatures to decide to either accept or reject the block.

The protocol assumes all validating nodes form a validator set, which is shared among all nodes participating in the consensus protocol. The assumption is guaranteed by logging inclusions and

Figure 4. Fork resolution protocol

exclusions in blocks. To build the validator set, a node builds the chain to the current tip (last block), and upon verifying each block, executes the validator state transition function to reconstruct the validator set. The state transaction function simply stores changes to the membership of the validator set. Nodes that want to participate in the consensus gossip their signed inclusion request, once included into a block, they are considered in the validator set by all nodes simultaneously, and can begin participating by role self-election. Nodes are excluded from the validator set when they are elected to a role of *Block Producer*, and fail to deliver the candidate block to the committee in time. The Committee will then vote for a skip block, which includes only the exclusion of the *Block Producer*. In a permissioned setting, this is considered sufficient to evaluate future failures in case a node is faulty. The node can rejoin the validator set at any time by gossiping an inclusion request. We define the consensus protocol more formally in Algorithm 3.

Algorithm 2 Role election

```
Input: Slot, ValidatorSet
Output: Roles[]
    $Slot_{seed} \Leftarrow VDF(chain_{(slot-1)}.hash, chain_{(slot-1)}.difficulty)$
    $ValidatorSet \Leftarrow Shuffle(ValidatorSet, Slot_{seed})$
    $Roles['blockProducer'] \Leftarrow ValidatorSet.subset(0,1)$
    $Roles['committee'] \Leftarrow ValidatorSet.subset(1, committeeSize)$
    $Roles['validator'] \Leftarrow ValidatorSet.subset(committeeSize, ValidatorSet.size)$
```

3.5. Security and fault tolerance considerations

Fault tolerance is an important property of the system. The system must guarantee the liveliness of applications running at any given time. Hence the risk of accidental forks (a split in the blockchain) must be examined. In a permissioned setting, forks are accidental and are a product of node failures or message propagation delays. We provide various scenarios of forks, and show how the fork choice rule addresses them.

1. The proposed block b for the current slot s is not propagated to all committee members in time C. A forked subset $C_f \subset C$ votes, and includes a skip block sb for slot s.
   (i) when $\frac{|C|}{2} > C_f$, block b will pass the majority vote, and the tip of the chain is b. However, C_f tip is sb. In this case C_f will produce different role assignments, and attempt to build on sb. Even if the block producer $bp \in C_f$, a majority vote cannot pass as $\frac{|C|}{2} > C_f$. Therefore C_f will add another sb. Eventually, the real block will reach the forked nodes, and due to a hash mismatch, nodes will initiate the fork resolution protocol.
   (ii) when $\frac{|C|}{2} < C_f$, block b will **not** pass the majority vote, and the tip of the chain is sb. No fork will occur.
2. Alternatively, attestations for b can be aggregated in time, but b fails to propagate to all committee members in time. A subset of committee members may then assume the block producer experienced a fault, and start gossiping sb. A network partition in the validator set occurs due to a race condition. However, eventually b will reach nodes with the tip sb. Due to a hash mismatch, they will initiate the fork resolution protocol.

Fig. 4 illustrates how fork resolution works. At height 2, two different blocks are proposed and accepted. both reference the correct previous block hash at which all nodes agreed on the same block. However, any blocks after height 2, will have a different previous block hash. Eventually, one of the chains has to be dismissed. For each of the aforementioned cases where a fork can occur, this eventually happens. In case of disconnect or high latency, the network eventually reaches higher connectivity

Algorithm 3 Consensus

```
Input: Role[]
    switch ($Roles[nodeId]$)
    case blockProducer:
        $block.migrations \Leftarrow prepareMigrationPlan(containerStats)$
        $block.signature \Leftarrow sign(block)$
        $broadcast(block, committee)$
        $votes \Leftarrow await(\frac{slotTime}{3})$
        $block.votes \Leftarrow BLS.aggregate(votes)$
        if $hasMajority(block.votes)$ then
            $gossip(block)$
        else
            $skipBlock()$
        end if
    case committee:
        $candidateBlock \Leftarrow await(\frac{slotTime*2}{3})$
        if $candidateBlock == null$ then
            $skipBlock()$
        else
            $proof \Leftarrow verify(candidateBlock.proof)$
            $migrations \Leftarrow verify(candidateBlock.migrationPlan)$
            $signature \Leftarrow verify(candidateBlock.signature)$
            if $(proof \& migrations \& signature)$ then
                $send(vote, blockProducer)$
            else
                $skipBlock()$
            end if
        end if
    case validator:
        $block \Leftarrow await(slotTime)$
        if $block == null$ then
            $skipBlock()$
        else
            $proof \Leftarrow verify(block.proof)$
            $migrations \Leftarrow verify(block.migrationPlan)$
            $signature \Leftarrow verify(block.signature)$
            $votes \Leftarrow verify(block.votes)$
            if $(proof \& migrations \& signature) \& votes)$ then
                $chain \Leftarrow block$
            end if
        end if
    end switch
```

as peers build new connections. Moreover, for each slot, nodes take up new roles in the consensus protocol, and the likelihood of effected nodes to maintain the same roles decreases exponentially.

In Nakamoto-style [21] consensus algorithms, the fork choice rule states that the longest chain (most proof of work) is the correct chain. However, a vote/role based consensus reduces variance in block time and the forked chain can have an identical block height. Instead, when a node cannot add a new valid block due to a mismatch of previous block hashes it backtracks to rechecking the attestations for each block down to the forked block, and afterwards rebuilds the chain including the blocks by following the chain with most cumulative attestations. Note that in order for a node to receive a valid block with a different previous block hash, the network partitions/high latency had to be resolved for the node to receive the alternative chain.

Another aspect of forks is the fault tolerance related to applications running in the system. Every block includes a migration plan, and in case of a fork, two or more migration plans are created and accepted by two disjoint sets of nodes. A migration plan will include all applications, which guarantees liveliness and variability of computation. Moreover, in case of a chain split, both sets of nodes (A, B) will execute the their respective plans which can unfold in the following two ways:

1. An application λ is planned to migrate from a node in A, to a node in set B or vice versa.
2. An application λ is planned to migrate to another node within sets A or B
3. An application λ does not need to migrate in either chain.

In order for an application to migrate from one node to another, a direct connection between the nodes must be established where one node sends a compressed version of the container to the other. To execute this, both the origin and destination node must agree and run the migration protocol. In case a fork occurred due to high network latency or complete disconnection between the two sets, the migration protocol will attempt to communicate between the sets, resolving the fork as shown in Fig. 4 as long as communication is possible. Whenever a migration plan requires an application to migrate between two conflicting chains, it forces a fork resolution. Moreover, in such cases, only one application is run at the same time. However, in the event application λ is planned to migrate within its originating chain, the migration will not force the network to reconnect. This could be considered a hard fork as there is no connection between the two networks, and results in separate instances of the network. The final example is when application λ is not required to migrate in which case one instance remains running and the system is not affected. The only example where serious faults might occur is when the network is well connected and forks because two nodes drew a winning ticket (same number). However, the migration algorithm is deterministic and the input is in the previous block. This means both block producers will produce the same migration plan even though the block hashes will be different. Although there will be two conflicting chains, the migration plan will be the same.

4. Evaluation and empirical results

To analyze the performance of our implementation, a node was selected to perform logging operations about the state into a time series database. The test environment was built using Docker Swarm to create a cluster. The cluster is comprised of 8 nodes, each with a 16-core (32 thread) Ryzen Zhreadripper CPU, and 32GB of RAM. The cluster nodes form an overlay network where latencies are almost non existent. Hence, artificial latency was added on individual UDP packets transmitted to create a more realistic environment. To deploy the network, a Docker service was created that runs the containerized node software across the cluster balancing the load across nodes. Each test-net was started with a boostrap setup, whereby the first node is considered the bootstrapping node, and it's public key, and IP address is known to all other nodes. The Docker service starts a new node every 10 seconds to avoid unrealistic network congestion. Each node was limited to 1 CPU core, and 256MB of RAM which exceeds the requirements for running the protocol. Applications were also submitted as Docker images and were able to execute by having each node run a Docker daemon inside their Docker container instance. This two level abstraction allowed the test-net to separate the node resources, and application resources from the host. Figure 5 outlines the architecture used by the cluster.

Figure 5. Cluster architecture

4.1. *Consensus Layer*

To verify that VDF based consensus provides good decentralized randomness, we analyze the distribution of assigned roles. Figure 6 shows the frequency nodes were elected into individual roles. Additionally, since container resource consumption statistics are propagated through a decentralized k-means clustering, cluster representatives are also shown. We observe that nodes have been elected into all roles, while there is some variance in the block producer role, the sample size is only 1000 slots in which only one node is selected as block producer for each slot; a more uniform distribution is expected with a larger sample size. Additionally, nodes joined the network gradually, which also effected the distribution.

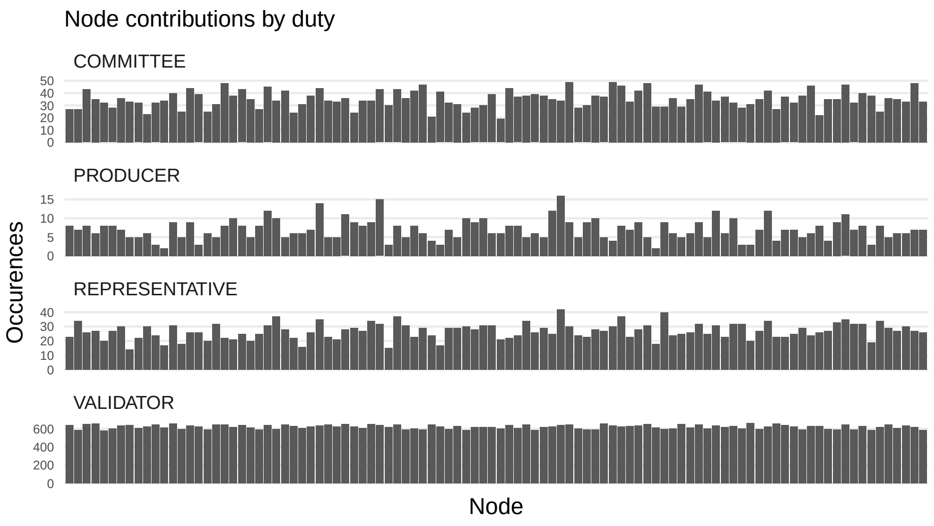

Figure 6. Distribution of roles across all participating nodes.

To validate the scalability of the consensus layer, we examined a network of 1000 nodes with a committee size of 256 nodes, and a target block time of 16 seconds. Figure 7 shows the block times, and number of votes per block that were successfully aggregated within the time window for the given slot. We observe that all proposed blocks were accepted as the majority vote threshold was surpassed, and no skip blocks were produced despite the low block time, and size of the committee. Moreover, we observe almost no variance in block time indicating that the system had no issue propagating messages.

4.2. *"Orchestration and Migration"*

One of the most important features of the system is the ability to migrate applications in a decentralized, transparent, and verifiable way. The decentralized orchestrator aims to distribute load across the network evenly by migrating applications away from nodes with heavy load to those with resources available. To test the performance and efficiency of migrations, we consider the worst case scenario in which all applications were submitted to one node.

Figure 8 illustrates the CPU load of nodes across the last 750 slots because nodes join the network gradually, and affect the early distribution, which skews the observations. We observe that the orchestrator migrated applications away from nodes with high CPU consumption to nodes with more available resources. This resulted in a gradual decline of the mean CPU load across the network.

In Figure 9 we compare both migration times of both test-nets to evaluate the feasibility and performance of CRIU enabled migrations. We break down a migration into 3 steps: *Save*, *Transmit*, and *Resume*. For standard migrations, saving requires pausing the running container, exporting, and

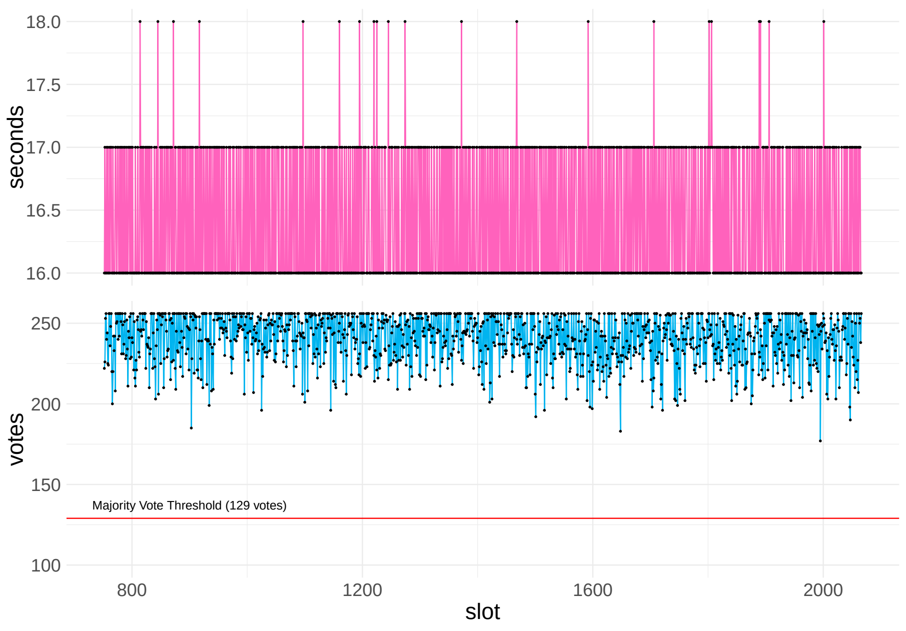

Figure 7. Committee vote aggregation, and block times in a network of 1000 nodes, and 256 committee members.

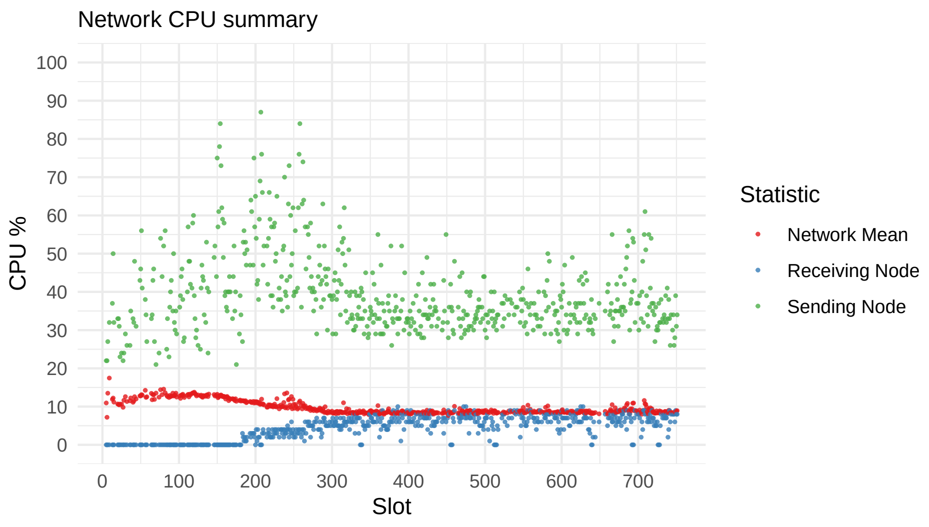

Figure 8. CPU load distribution of the entire network over the last 750 slots.

compressing it. In CRIU, the container is also paused but instead of exporting it, only the state is extracted and compressed. After transmission, the receiving node must resume the container. In standard migrations, the container is uncompressed, and resumed, while using CRIU, a new container from the same base image is created, and the uncompressed state is injected into it.

Using CRIU, the payload for transmission is much smaller, and hence transmission time is greatly improved over standard. Additionally, compression is a CPU intensive task. Compressing and decompressing only the state of an application instead of the entire container is considerably faster. The median uncompressed exported state of the application using standard migration was 142.2 MB. Using CRIU, the median size of the uncompressed state was 15.2MB. The spikes in standard migrations can be attributed to lack of resources as nodes under heavy stress from running other applications lack the resources needed to perform the compression promptly. Table 2 provides a statistical summary of the observed times in milliseconds. We observe that CRIU enabled migrations are not only faster but also produce more consistent migration times. This can be observed by the considerably lower standard deviation in Table 2.

Type	Segment	Min.	Max.	Med.	Mean	SD
CRIU	Resume	2366	10012	3259	3540	1166
CRIU	Save	1975	8368	2701	3126	1111
CRIU	Transmit	46	833	79	88	61
Standard	Resume	1449	34337	7414	9550	6637
Standard	Save	4010	49080	11231	12875	6942
Standard	Transmit	506	15007	1624	2047	1467

Table 2. Summary of migration times in milliseconds.

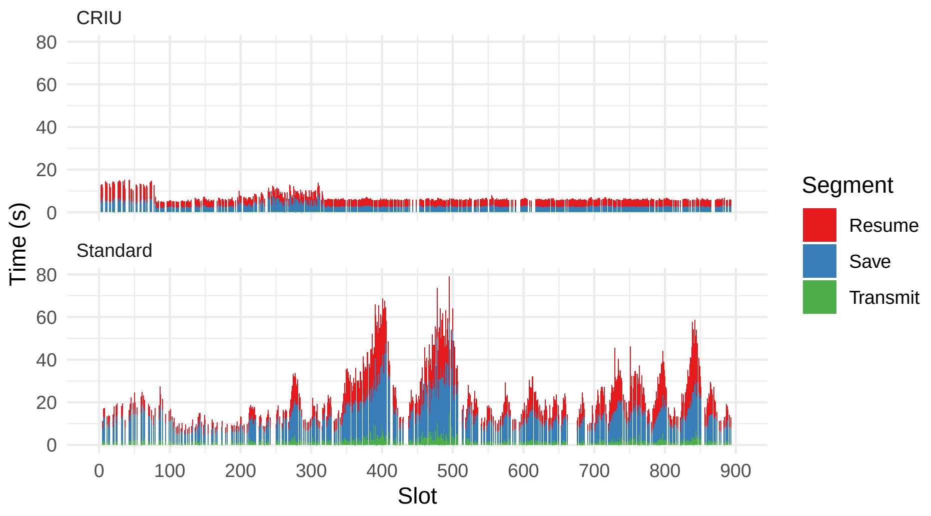

Figure 9. Migration timing comparison between standard and CRIU enabled migrations.

Another way to visualise the dynamics of the system is shown in Figure 10. Each polarized bar chart shows nodes on x axis. Stacked bars are used to illustrate the number of applications running on the node, and their respective CPU consumption in %. We observe that, initially, the application distribution was uneven with a very high CPU load on one node. This is the result of submitting incoming applications to one node. Over time, the system is able to evenly distribute applications across the network.

4.3. Network Clustering

The performance of the orchestrator heavily depends on the propagation speed of resource allocation from all validators. In a clustered network, *validators* report their resources to their cluster *representatives*, which finally send an aggregated report to the block producer. To avoid a potential attack vector on the clustering, the network topology changes every slot. Figure 11 shows the time needed to deliver the resource reports to *representatives*, and finally the *producer*. We observe that in the first few minutes while the nodes are joining the network at a high frequency the propagation times are noticeably slower (still well within the $\frac{1}{block_t ime}$) but stabilize quickly even with networks of 1000 nodes.

5. Conclusions

In this paper, we introduced a decentralized architecture capable of run-time application migration for large scale deployments of peer-to-peer IoT sensor networks. We make three key contributions,

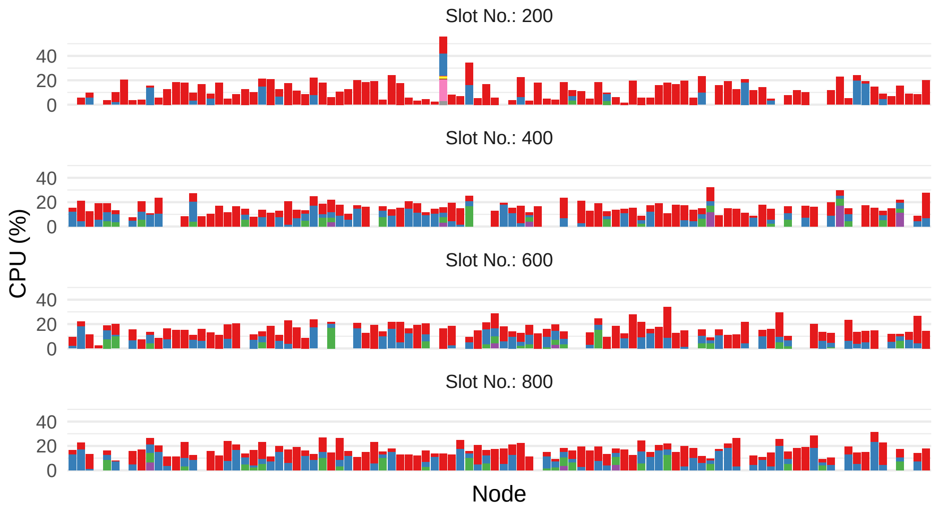

Figure 10. Discrete time visualisation of applications and their CPU load utilization on participating nodes. Colours indicate individual containers active on each node, and are not necessarily the same container on different nodes.

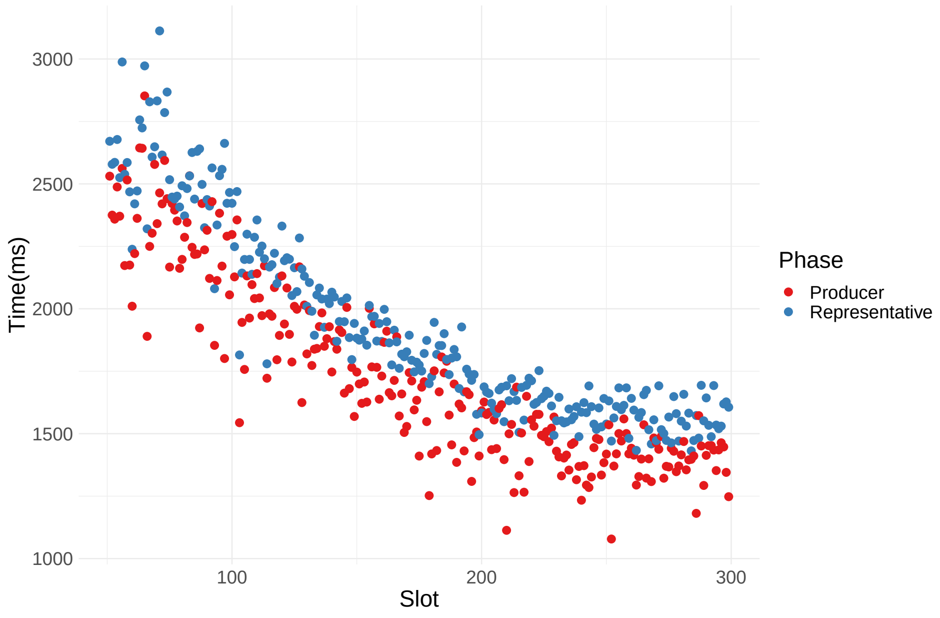

Figure 11. Time distribution of resource propagation int two phases. Initially, validators submit their resource statistics to their respective cluster representatives (*To representative*). After, cluster representatives send collected reports to the block producer (*To Producer*). There were a total of 50 clusters created each slot within a target slot time of 16 seconds, which sets the upper bound for resource propagation at 5.4 seconds.

namely a scalable consensus layer, an efficient, secure and dynamic topology, and a decentralized orchestrator capable of low latency real time application migrations.

We evaluate each contribution by performing empirical tests with our reference implementation of the protocol. Additionally, we improve migration times by implementing CRIU, an experimental feature of Docker that allows the system to migrate an application's state without effecting its run-time. Using CRIU enabled migrations, we observe considerable reduction (nearly 10-fold) and improved consistency in migration times.

The results of our experiments show that distributed consensus and application management is possible at run-time, thus opening the door to several improvements towards self-managing IoT platforms. The increase in network usage and CPU load has shown to be acceptable when taking into account the scalability, fault tolerance, transparency, and absence of a SPOF that our solution brings. Importantly, we have shown that blockchain overhead is a negligible aspect of the actual cost of application migration as the system is able to finalize blocks with slot times as low as 5 seconds while maintaining higher decentralization then existing platforms such as Multichain, which uses a variation of practical byzantine fault tolerance (PBFT) consensus, and Hyperledger Fabric [41] which uses Raft [39].

As future work, we will explore the limits of our solution with respect to network instability (devices entering and leaving the network) and explore solutions to reduce the required computational power while maintaining optimal application management. Moreover, the algorithm governing the decentralized orchestrator will be extended to allow applications to submit migration policies the orchestrator will respect. As future work, more efficient orchestration algorithms should be explored with emphasis given on performing multiple migrations in the same slot with a non-cycle constraint.

Further, geo-sharding the network must be explored. In a geo-sharded network, nodes participating are assigned into shards based on their geographical location. A weaker consensus within a shard can speed up the state transition by periodically snapshotting sharded states into the main chain. This will enable applications to specify more complex migration policies by limiting a geographical area within which the application may run (geo-fencing). Moreover, a geographically aware system can perform better migrations by migrating applications closer to clients in order to improve network latency. Using ereasure coding, the storage requirements of individual nodes is greatly reduced [42] as full replication is not needed.

Author Contributions: Conceptualization, A.T., J.V. and M.M.; methodology: A.T., J.V., M.M. and M.B.; software, A.T.; validation, A.T., J.V., M.M. and M.B.; formal analysis, A.T., J.V., M.M. and M.B.; investigation, A.T., J.V., and M.M.; funding acquisition and resources, J.V., M.B. and M.M.; data curation, A.T. and M.B; writing–original draft preparation, A.T.; writing–review and editing, A.T., J.V., M.M. and M.B.; visualization, A.T. and M.B.

Funding: This research was funded by H2020 grant numbers 739574 and 857188 and by the Slovenian Research Agency (ARRS) grant number J2-2504.

Acknowledgments: The authors gratefully acknowledge the European Commission for funding the InnoRenew CoE project (H2020 Grant Agreement #739574) and the PHArA-ON project (H2020 Grant Agreement #857188) and the Republic of Slovenia (Investment funding of the Republic of Slovenia and the European Union of the European Regional Development Fund) as well as the Slovenian Research Agency (ARRS) for supporting project number J2-2504.

Conflicts of Interest: "The authors declare no conflict of interest.'The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results".

References

1. Jaeger, P.T.; Lin, J.; Grimes, J.M.; Simmons, S.N. Where is the cloud? Geography, economics, environment, and jurisdiction in cloud computing. *First Monday* **2009**, *14*.
2. Khalid, Z.; Fisal, N.; Rozaini, M. A Survey of Middleware for Sensor and Network Virtualization. *Sensors* **2014**, *14*, 24046–24097. doi:10.3390/s141224046.
3. Garcia Lopez, P.; Montresor, A.; Epema, D.; Datta, A.; Higashino, T.; Iamnitchi, A.; Barcellos, M.; Felber, P.; Riviere, E. Edge-centric Computing: Vision and Challenges. *SIGCOMM Comput. Commun. Rev.* **2015**, *45*, 37–42. doi:10.1145/2831347.2831354.
4. Hoebeke, J.; Moerman, I.; Dhoedt, B.; Demeester, P. An overview of mobile ad hoc networks: applications and challenges. *Journal-Communications Network* **2004**, *3*, 60–66.
5. Sha, K.; Wei, W.; Yang, T.A.; Wang, Z.; Shi, W. On security challenges and open issues in Internet of Things. *Future Generation Computer Systems* **2018**, *83*, 326–337.
6. De Souza, L.M.S.; Vogt, H.; Beigl, M. A survey on fault tolerance in wireless sensor networks. *Interner Bericht. Fakultät für Informatik, Universität Karlsruhe* **2007**.

7. Padmavathi, D.G.; Shanmugapriya, M.; others. A survey of attacks, security mechanisms and challenges in wireless sensor networks. *arXiv preprint arXiv:0909.0576* **2009**.
8. Taherizadeh, S.; Jones, A.C.; Taylor, I.; Zhao, Z.; Stankovski, V. Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review. *Journal of Systems and Software* **2018**, *136*, 19–38.
9. Östberg, P.O.; Byrne, J.; Casari, P.; Eardley, P.; Anta, A.F.; Forsman, J.; Kennedy, J.; Le Duc, T.; Marino, M.N.; Loomba, R.; others. Reliable capacity provisioning for distributed cloud/edge/fog computing applications. 2017 European conference on networks and communications (EuCNC). IEEE, 2017, pp. 1–6.
10. Le Duc, T.; Oestberg, P.O. Application, Workload, and Infrastructure Models for Virtualized Content Delivery Networks Deployed in Edge Computing Environments. 2018 27th International Conference on Computer Communication and Networks (ICCCN). IEEE, 2018, pp. 1–7.
11. Diallo, M.H.; August, M.; Hallman, R.; Kline, M.; Slayback, S.M.; Graves, C. AutoMigrate: a framework for developing intelligent, self-managing cloud services with maximum availability. *Cluster Computing* **2017**, *20*, 1995–2012.
12. Peltz, C. Web services orchestration and choreography. *Computer* **2003**, *36*, 46–52.
13. Hightower, K.; Burns, B.; Beda, J. *Kubernetes: Up and Running: Dive Into the Future of Infrastructure*; " O'Reilly Media, Inc.", 2017.
14. Mercl, L.; Pavlik, J. The comparison of container orchestrators. Third International Congress on Information and Communication Technology. Springer, 2019, pp. 677–685.
15. Acuña, P. Amazon EC2 Container Service. In *Deploying Rails with Docker, Kubernetes and ECS*; Springer, 2016; pp. 69–98.
16. Rathi, V.K.; Chaudhary, V.; Rajput, N.K.; Ahuja, B.; Jaiswal, A.K.; Gupta, D.; Elhoseny, M.; Hammoudeh, M. A Blockchain-Enabled Multi Domain Edge Computing Orchestrator. *IEEE Internet of Things Magazine* **2020**, *3*, 30–36. doi:10.1109/IOTM.0001.1900089.
17. Savi, M.; Santoro, D.; Di Meo, K.; Pizzolli, D.; Pincheira, M.; Giaffreda, R.; Cretti, S.; Kum, S.w.; Siracusa, D. A Blockchain-based Brokerage Platform for Fog Computing Resource Federation. 2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN), 2020, pp. 147–149. doi:10.1109/ICIN48450.2020.9059337.
18. Pires, A.; Simão, J.; Veiga, L. Distributed and Decentralized Orchestration of Containers on Edge Clouds. *Journal of Grid Computing* **2021**, *19*, 1–20.
19. Mazzoni, E.; Arezzini, S.; Boccali, T.; Ciampa, A.; Coscetti, S.; Bonacorsi, D. Docker experience at infn-pisa grid data center. Journal of Physics: Conference Series. IOP Publishing, 2015, Vol. 664, p. 022029.
20. Buterin, V.; others. Ethereum white paper. *GitHub repository* **2013**, *1*, 22–23.
21. Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system, 2009.
22. Schäffer, M.; di Angelo, M.; Salzer, G. Performance and Scalability of Private Ethereum Blockchains. Business Process Management: Blockchain and Central and Eastern Europe Forum; Di Ciccio, C.; Gabryelczyk, R.; García-Bañuelos, L.; Hernaus, T.; Hull, R.; Indihar Štemberger, M.; Kő, A.; Staples, M., Eds.; Springer International Publishing: Cham, 2019; pp. 103–118.
23. Thakkar, P.; Nathan, S.; Viswanathan, B. Performance benchmarking and optimizing hyperledger fabric blockchain platform. 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). IEEE, 2018, pp. 264–276.
24. Ismailisufi, A.; Popović, T.; Gligorić, N.; Radonjic, S.; Šandi, S. A private blockchain implementation using multichain open source platform. 2020 24th International Conference on Information Technology (IT). IEEE, 2020, pp. 1–4.
25. Yakovenko, A. Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper* **2018**.
26. Maior, H.A.; Rao, S. A self-governing, decentralized, extensible Internet of Things to share electrical power efficiently. 2014 IEEE International Conference on Automation Science and Engineering (CASE). IEEE, 2014, pp. 37–43.
27. Higgins, N.; Vyatkin, V.; Nair, N.K.C.; Schwarz, K. Distributed power system automation with IEC 61850, IEC 61499, and intelligent control. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* **2011**, *41*, 81–92.
28. Suzdalenko, A.; Galkin, I. Instantaneous, short-term and predictive long-term power balancing techniques in intelligent distribution grids. Doctoral Conference on Computing, Electrical and Industrial Systems. Springer, 2013, pp. 343–350.

29. Niyato, D.; Xiao, L.; Wang, P. Machine-to-machine communications for home energy management system in smart grid. *IEEE Communications Magazine* **2011**, *49*, 53–59. doi:10.1109/MCOM.2011.5741146.
30. Bragard, Q.; Ventresque, A.; Murphy, L. Self-balancing decentralized distributed platform for urban traffic simulation. *IEEE Transactions on Intelligent Transportation Systems* **2017**, *18*, 1190–1197.
31. Al-Madani, B.M.; Shahra, E.Q. An Energy Aware Plateform for IoT Indoor Tracking Based on RTPS. *Procedia computer science* **2018**, *130*, 188–195.
32. Teh, P.L.; Ghani, A.A.A.; Chan Yu Huang. Survey on application tools of Really Simple Syndication (RSS): A case study at Klang Valley. 2008 International Symposium on Information Technology, 2008, Vol. 3, pp. 1–8. doi:10.1109/ITSIM.2008.4631980.
33. Samaniego, M.; Deters, R. Using blockchain to push software-defined IoT components onto edge hosts. Proceedings of the International Conference on Big Data and Advanced Wireless Technologies. ACM, 2016, p. 58.
34. Tošić, A.; Vičič, J.; Mrissa, M. A Blockchain-based Decentralized Self-balancing Architecture for the Web of Things. European Conference on Advances in Databases and Information Systems. Springer, 2019, pp. 325–336.
35. Bozyigit, M.; Wasiq, M. User-level process checkpoint and restore for migration. *ACM SIGOPS Operating Systems Review* **2001**, *35*, 86–96.
36. Dwork, C.; Naor, M. Pricing via Processing or Combatting Junk Mail. Advances in Cryptology — CRYPTO' 92; Brickell, E.F., Ed.; Springer Berlin Heidelberg: Berlin, Heidelberg, 1993; pp. 139–147.
37. Castro, M.; Liskov, B.; others. Practical byzantine fault tolerance. OSDI, 1999, Vol. 99, pp. 173–186.
38. Chen, L.; Xu, L.; Shah, N.; Gao, Z.; Lu, Y.; Shi, W. On security analysis of proof-of-elapsed-time (poet). International Symposium on Stabilization, Safety, and Security of Distributed Systems. Springer, 2017, pp. 282–297.
39. Ongaro, D.; Ousterhout, J. In search of an understandable consensus algorithm. 2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14), 2014, pp. 305–319.
40. Boneh, D.; Bonneau, J.; Bünz, B.; Fisch, B. Verifiable delay functions. Annual International Cryptology Conference. Springer, 2018, pp. 757–788.
41. Androulaki, E.; Barger, A.; Bortnikov, V.; Cachin, C.; Christidis, K.; De Caro, A.; Enyeart, D.; Ferris, C.; Laventman, G.; Manevich, Y.; others. Hyperledger fabric: a distributed operating system for permissioned blockchains. Proceedings of the thirteenth EuroSys conference, 2018, pp. 1–15.
42. Perard, D.; Lacan, J.; Bachy, Y.; Detchart, J. Erasure code-based low storage blockchain node. 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). IEEE, 2018, pp. 1622–1627.