Article

A Blockchain-based Edge Computing Architecture for the Internet of Things

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

[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, Slovenija; aleksandar.tosic@innorenew.eu
[4] InnoRenew CoE; Livade 6 6310 Izola, Slovenija; mike.burnard@innorenew.eu
[5] InnoRenew CoE; Livade 6 6310 Izola, Slovenija; michael.mrissa@innorenew.eu
[*] 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. To address these concerns, we introduce a decentralized architecture tailored for large scale deployments of peer-to-peer IoT sensor networks and capable of run-time application migration. The solution combines a blockchain consensus algorithm and verifiable random functions to ensure scalability, fault tolerance, transparency, and no single point of failure. We build on our previously presented theoretical simulations [1] with many protocol improvements and an implementation tested in a use case related to monitoring a Slovenian cultural heritage building located in Bled, Slovenia.

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 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. As a matter of fact, 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 [2]. 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. These edge devices are heterogeneous and existing IoT platforms remain isolated and do not 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

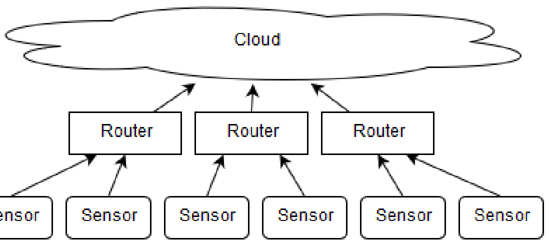

Figure 1. Standard sensor network architecture.

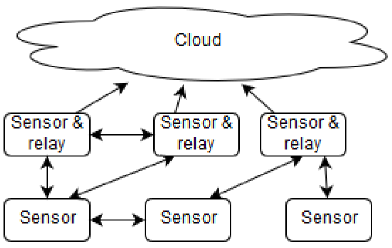

Figure 2. Mesh sensor network architecture.

of edge devices, which requires applications to migrate at run-time from an overloaded device to a more available one. Currently, there is not a 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 middle layer infrastructure that enables access to the cloud (Fig. 1).

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 Internet (Fig. 2).

The need for edge computing is well illustrated with all scenarios related to ad-hoc networks [4], particularly with peer-to-peer wireless sensor networks. The paradigm 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 purposed 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 would send their measurements to the cloud where the computation would occur. However, issues such as poor network coverage, frequent disconnection, cost of infrastructural investment, inadequate dependability, and security concerns remain unaddressed. Edge computing solutions aim at reducing network latency, increasing fault tolerance, dependability and security, and at reducing 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.). The decision making process for resource allocation is made by the decentralized orchestrator implemented as a blockchain protocol. It has high fault tolerance, provides full transparency, reduced network infrastructure cost, and no single point of failure.

2. Background knowledge and related work

No widely-used cloud monitoring tool for edge computing fully addresses monitoring as some requirements are not met by any existing solution, as explained in a recent study [5]. That study also stresses the importance of other system levels to monitor including container, end-to-end network quality, and application levels. Finally, the most critical unmet monitoring challenges according to the same study 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. The EU project RECAP [6] presents a vision of the next generation of intelligent, self-managed, and self-remediated (monitor and relocate resources to achieve Quality of Service - QoS) cloud computing systems. This vision is generally aligned with our proposal, and in this paper we deliver a pilot implementation. One of the deliverables of the project [7] describes models intended to be integrated in network topology-aware application orchestration and resource management systems from an edge computing perspective. AutoMigrate [8] 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. AutoMigrate relies on a central service to orchestrate migrations. Our decentralized implementation eliminates the SPOF issue.

2.1. Orchestration solutions for edge computing

We have identified the most successful solutions to be Kubernetes [9], the most used and most feature-rich orchestration tool [10], Docker Swarm[1], AWS Elastic Container Service (AWS ECS) [11], DC/OS[2], and Nomad[3]. All presented architectures have a common flaw that we are trying to address in our paper: the Single Point Of Failure (SPOF) problem. The flaw is characterised by a single orchestration entity. Most solutions also lack support for edge devices. All identified orchestration solutions rely on a primary/replica model. The main service selects the applications that need to be reallocated according to a selected optimization algorithm. 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 described open problems.

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 [13] 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. The algorithm for migrations:

- pause a container,

[1] https://github.com/docker/swarm
[2] https://dcos.io/
[3] https://www.hashicorp.com/products/nomad

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

Additionally, we implemented the migrations using Checkpoint/Restore In Userspace, or CRIU. CRIU is an experimental feature available in docker. We present our comparison between both migration strategies.

2.3. 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. [14] 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. [15] propose a distributed IoT approach for electrical power demand management.
- Suzdalenko and Galkin [16] extend the approach by Higgins et al. [15] by allowing users to individually and in run-time join and part the environment.
- Niyato et al. [17] propose a system that addresses the home energy management system where machines communicate directly among themselves.
- dSUMO [18] address the bottleneck in synchronization by proposing a distributed and decentralized microscopic simulation (the focus is on the data throughput and not so much the fault tolerance, the throughput is increased using decentralised setting).
- Al-Madani et al. [19] address indoor localization utilizing Wireless Sensor Networks (WSNs) relaying on publish/subscribe messaging model. The results show that the Really Simple Syndication (RSS) [20] format achieves acceptable accuracy for multiple types of applications.

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

- other solutions mostly 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 even provision a framework to add more criteria in the future.
- the main contribution of this paper is the proposal of a data structure similar to blockchain and a consensus algorithms to manage application migration at run-time on the edge devices.

A related approach by Samaniego and Deters [21] 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.

3. Verifiable decentralized architecture

In this section, we provide a general description of our architecture[4] 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. The decentralized orchestrator would then be able to choreograph the execution of the application, and make changes on 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. 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.

[4] Preliminary version available at https://arxiv.org/abs/1904.09595

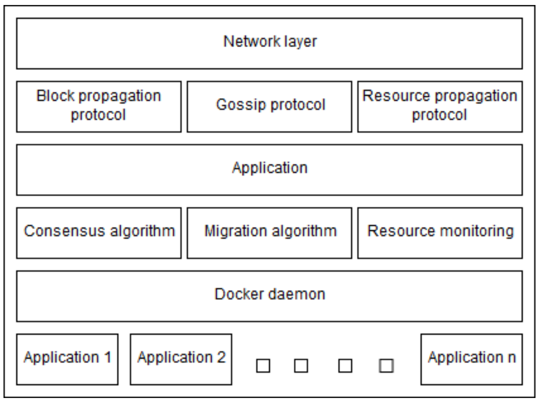

Figure 3. General overview of a node architecture.

3.1. Overview of node architecture

Our node architecture 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[5], 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 gossip protocol to spread usage state of resources (currently CPU, RAM, disk usage, and network utilization of docker containers) over the network.
- Consensus protocol: ensures all nodes reach consensus in a decentralized way (presented below).
- Migration algorithm: guarantees that a deterministic migration strategy is reached whenever needed. This algorithm is executed at each block by the block producer. The output 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 Docker API to monitor the state, and resource allocations of the hosting device and the applications running on it.

Fig. 3 shows the internal structure of a node and the node application we developed that is containerized in Docker.

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 Chapter 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

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

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.

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 fast. Using the the gossiping 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 responsible for requesting resource utilization statistic 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 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 the state, including all in-memory data, and transferring it to the destination node where it is restored. 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 accidental 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.

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
```

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. Due to the increased interest in blockchain technology in recent years, many consensus algorithms that build upon basic proof of work[6] were introduced. However, most algorithms used in permission-less 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 (financial) advantage (i.e., the nothing at stake problem). Miners in Bitcoin are incentivised by receiving rewards for mining and supporting the network, validators in proof of stake are incentivised to validate transaction by being rewarded transaction fees, and penalized (stake taken away from them) if they are malicious. However, in permissioned networks, where there is usually no monetary value, the aforementioned consensus algorithms are not appropriate. A known family of consensus algorithms for permissioned networks are voting schemes for leader election such as PBFT [23], Proof of Elapsed Time (PoET) [24] or RAFT [25]. However, these algorithms require multiple messages to be sent through the network in order to commit a change. We propose a leader election consensus algorithm for permissioned distributed networks that only requires one series of messages in order to commit to a change. The algorithm is based on a lottery ticket draw that is universally verifiable. A recent paper proposed construction of verifiable random functions (VRF) [26], where a node can compute a VRF f given an input value x, and a private key SK to generate a random value y and a universally verifiable proof p that y was indeed the output of f using the public key PK without revealing SK.

A lottery-based leader election for block n can be implemented in the following few steps:

1. Assume a block height (canonical id) of n and a node's most recent block is also n
2. A node draws a ticket (compute F) using the current block (n) hash a seed
3. A node forges block $n+1$ with the proposed migration plan
4. A node sends the proposed block coupled with y (ticket), and p (proof)
5. A node receiving many proposed blocks will include the one with the smallest y, and a valid p
6. The block $n+1$ reaches finality when the node receives block $n+2$

[6] Proof of work [22] 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.

Such a lottery scheme (leader is the node with the smallest y) is quite efficient. However, malicious nodes can compute the VRF many times and commit to the lowest number since the lottery rules are known in advance. In order to secure the network, all nodes would have to draw many tickets, which would be analogous to proof of work where nodes with the most computational power increase their odds of winning the lottery by drawing more tickets.

To circumvent this problem we use the recently published [27] construction of Verifiable Delay Functions (VDF). 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 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 as outlined in Algorithm 2, namely *Block Producer*, *Validator*, *Committee Member*. 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 respective 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 sufficient number of votes are received, else 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 signatures (BLS) signatures, and sending to the block producer.
3. *Validator* nodes receive a new block, they 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 between all node participating in the consensus protocol. The assumption is guaranteed by logging inclusions and exclusions in blocks. To build the validator set, a node builds the chain to the current tip (last block), and upon verifying each block, execute 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 *blockProducer*, 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 *blockProducer*. In a permissioned setting, this is considered sufficient to elevate the future failures in case a node is faulty. The node can rejoin the validator set at any time by gossiping the inclusion request.

We define the consensus protocol more formally in Algorithm 3.4.

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.

Algorithm 2 Role election

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

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
```

Figure 4. Fork resolution protocol

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 forks fork resolution works. At height 2, two different blocks are proposed and accepted both referencing 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 as peer 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 [28] 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 latencies had to be resolved for the node to receive the alternative chain.

The other 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. Test implementation and empirical results

The test environment was built using docker swarm to create a cluster. The cluster is comprised of 8 nodes, each with a 32-core Ryzen 7 CPU, and 32GB of RAM. The cluster nodes form a local area 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 trusted setup, whereby the first node is considered trusted, 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. 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

To analyze the performance of our implementation, a node was selected to perform logging operations about the state into a time series database. We analyze the telemetry obtained from two test-nets, one using standard container migrations, and the other using CRIU. Both networks were tested with 100 nodes, and 200 applications running. To test the performance, and efficiency of migrations, we consider the worst case scenario in which all applications were submitted to one node.

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.

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. Figure 7 illustrates the CPU load of nodes across 750 slots. 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 8 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 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.

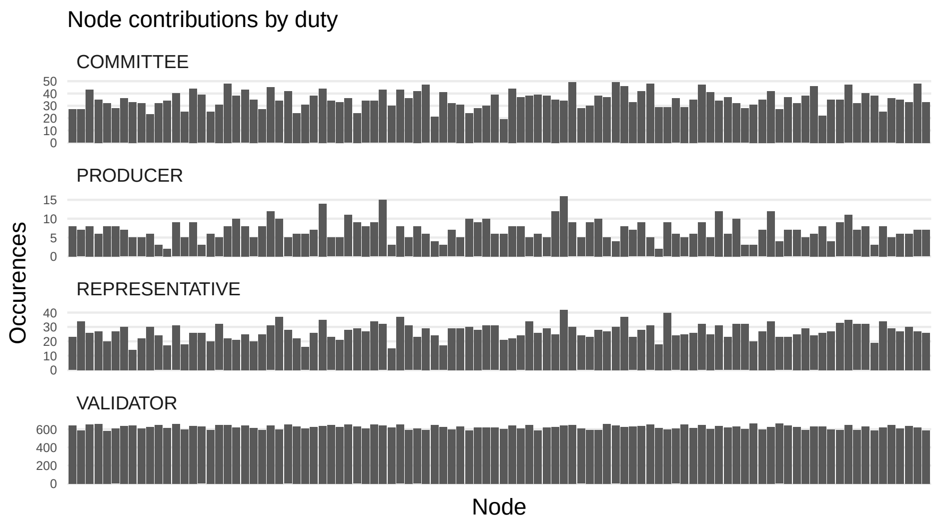

Figure 6. Distribution of roles across all participating nodes

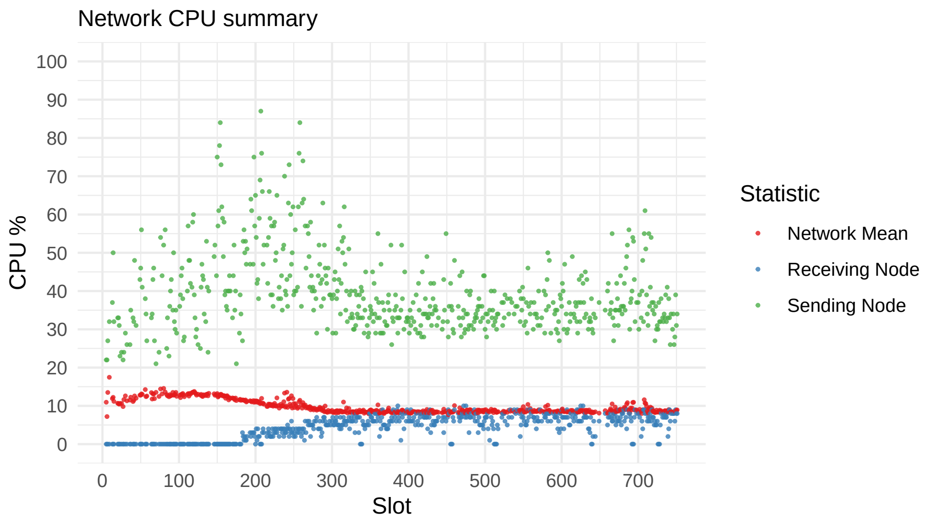

Figure 7. CPU load distribution of the entire network over 750 slots

Using CRIU, the 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 therefore also 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.

Another way to visualise the dynamics of the system is shown in Figure 9. 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 %. The inner polar bar chart is the state of the network at slot 200, while the rest are obtained by stepping up the slot by 200. We observe that, initially, the application distribution was uneven with a very high CPU load on one node. This is

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

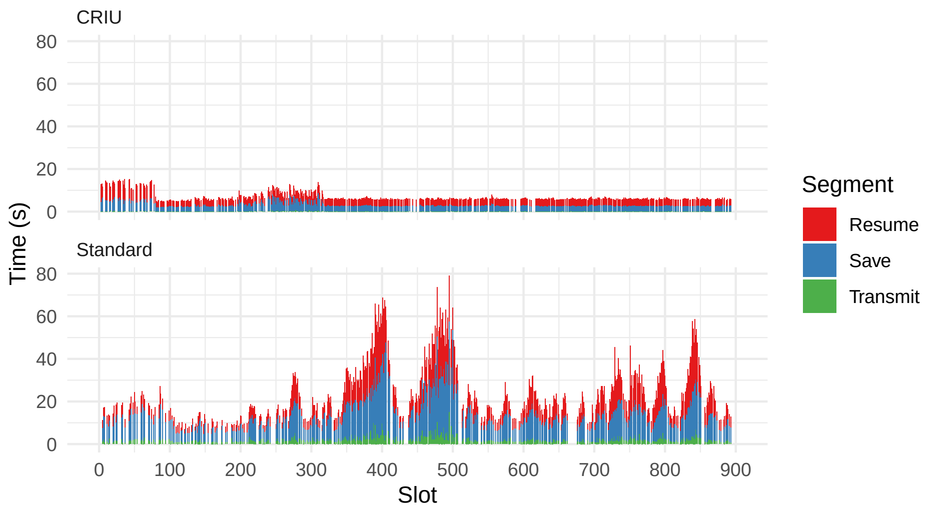

Figure 8. Migration timing compassion between standard and CRIU enabled migrations

the result of submitting incoming applications to one node. Over time, the system is able to evenly distribute applications across the network.

5. Conclusions

Currently, the growing number of edge devices and their increasing complexity, combined with containerization solutions, motivates a shift towards decentralized edge computing to support the management of IoT software. However, several properties such as scalability, fault tolerance, and transparency need to be guaranteed for this evolution to take place. Typically, these properties are provided through a supposedly reliable element in the network, thus creating a single point of failure.

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. It combines a blockchain consensus algorithm with verifiable delay functions to provide decentralized randomness, scalability, fault tolerance, transparency, and to remove any single point of failure. We demonstrate that both our blockchain and VDF implementations function on their own, and that application migrations function across a test network to balance the overall CPU load of the devices on the network. Additionally, we improve the migration times by implementing CRIU, an experimental feature of Docker that allows the system to migrate an applications state without effecting its run-time. Using CRIU enabled migrations, we observe considerable reduction 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 proven 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 future work, we will explore the limits of our solution with respect to

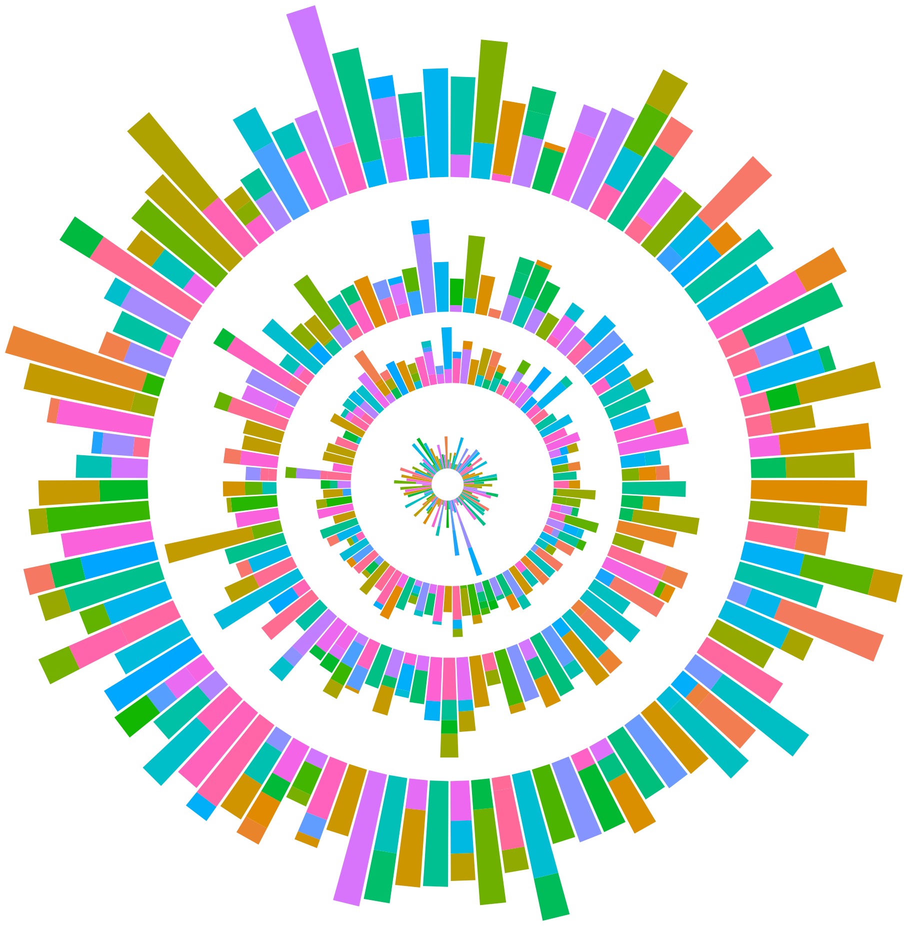

Figure 9. Discrete time visualisation of applications and their CPU load utilization on participating nodes

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.

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.

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.; 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 J2-9404 (C).

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 the project number J2-2504 (C).

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. 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.
2. 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*.
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. 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.
6. 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.
7. 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.
8. 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.
9. Hightower, K.; Burns, B.; Beda, J. *Kubernetes: Up and Running: Dive Into the Future of Infrastructure*; " O'Reilly Media, Inc.", 2017.
10. Mercl, L.; Pavlik, J. The comparison of container orchestrators. Third International Congress on Information and Communication Technology. Springer, 2019, pp. 677–685.
11. Acuña, P. Amazon EC2 Container Service. In *Deploying Rails with Docker, Kubernetes and ECS*; Springer, 2016; pp. 69–98.
12. Peltz, C. Web services orchestration and choreography. *Computer* **2003**, *36*, 46–52.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. 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.

19. 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.
20. 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.
21. 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.
22. 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.
23. Castro, M.; Liskov, B.; others. Practical Byzantine fault tolerance. OSDI, 1999, Vol. 99, pp. 173–186.
24. 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.
25. Ongaro, D.; Ousterhout, J. In search of an understandable consensus algorithm. 2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14), 2014, pp. 305–319.
26. Micali, S.; Rabin, M.; Vadhan, S. Verifiable random functions. 40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039). IEEE, 1999, pp. 120–130.
27. Boneh, D.; Bonneau, J.; Bünz, B.; Fisch, B. Verifiable delay functions. Annual International Cryptology Conference. Springer, 2018, pp. 757–788.
28. Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system, 2009.