UNIVERSIDAD INTERNACIONAL DE LA RIOJA

PROGRAMA DE DOCTORADO EN CIENCIAS DE LA COMPUTACIÓN

TESIS DOCTORAL

Towards Decentralized Service Orchestration for Heterogeneous Cloud Services

Memoria presentada por

Alberto Arias Maestro
para optar al grado de Doctor
por la Universidad Internacional de La Rioja

Dirigida por los Doctores:

Oscar Sanjuan Martinez
e
Vicente Garcia Diaz

Logroño, 2022

Acknowledgments

First and foremost, I want to thank my parents for their love and support throughout my life. Thank you both for making it all possible for me. Your hard work allowed me to chase after my dreams.

None of this work would have been possible without the collaboration and unconditional support, time, and words of encouragement and motivation from the many people I had the fortune to work with over the last 15 years. that I have been involved in this area of research.

I want to thank the supervising team for this thesis, Oscar Sanjuan Martinez, and Vincente Garcia Diaz. From the very beginning of this journey, Oscar was consistently an inspiration, guide, and facilitator. Vicente has been guiding, supporting, and encouraging me through each step of the way.

I also want to thank Ankur Teredasai for his encouragement and support throughout the evaluation and diffusion of results.

Lastly, thank you to my family for their unconditional support. My wife Josephine for always believing in me, and my twin daughters for sharing their wondrous curiosity.

Abstract

Over the last three decades, the broad adoption of the Internet triggered a shift toward a globally accessible and service-oriented application design. Hyper-scale services like e-commerce, search, social networking, IoT, and big data have driven the industry towards hyper-optimizing supporting infrastructure and management technologies. The need for accelerated enterprise digital transformation and connected application development have established the cloud as the de-facto solution for new application development.

I have been fortunate to be part of the Cloud industry for the last 15 years. From the emergence of the software-defined data center, virtualization of computing and network resources, multi-cloud orchestration, containers, and back hybrid cloud computing. The common theme across all of these technologies is the default implementation of centralized orchestration to manage and distribute resources to enable a wide range of scale requirements. As a director of Cloud Services at Google, I theorized that innovation and scale would grind to a halt if every policy change in the services network needed to be orchestrated and centrally coordinated.

This thesis examines decentralized architectures for cloud resource management by applying the recent practical advancements in blockchain and consensus technology. The proposed architecture provides the foundation for a fully distributed configuration management system that stores the global configuration in a blockchain structure and is spread across all the nodes in the network.

While there are dozens of blockchain platforms in existence. At the time of this writing, there is an ongoing transformation of the existing popular blockchain towards achieving the right balance between scalability, performance, and general-purpose utility. Layering, sharding, cross-chain smart contracts, and chain interoperability are some of the technical solutions still to be implemented broadly, making it difficult to predict the life expectancy of current blockchain platforms.

For this research, we evaluated many general-purpose blockchains yet decided to implement a custom-built solution for two reasons. The first reason is the additional management and resource overhead of running a complete smart contract platform on each network node. The second but no less important reason is the complexity of implementing a cloud resource management Domain Specific Language using expensive to run smart contracts based on imperative languages.

The solution researched is a custom-built chain. This architecture noticeably increases the system availability, including cases of network partitioning, without significantly impacting configuration consistency.

Keywords

Cloud Computing, Infrastructure as a Service, Containers, Orchestration, Distributed Systems, Domain Specific Language, Blockchain, Consensus Algorithms, Smart Contracts,

TABLE OF CONTENTS

TABLE INDEX

FIGURE INDEX

PART I: RESEARCH SCOPE AND OBJECTIVES

CHAPTER 1: INTRODUCTION

Over the last three decades, the broad adoption of the Internet triggered a shift toward a globally accessible and service-oriented application design [1]. Hyper-scale services [2] like e-commerce, search, social networking, IoT, and big data have driven the industry towards hyper-optimization of supporting infrastructure and management technologies [3]. The need for accelerated enterprise digital transformation and connected application development have established the cloud as the de-facto solution for new application development [4].

Today's cloud systems support thousands of mission-critical applications composed of multiple distributed heterogeneous components [5]. Maintaining end-to-end operational integrity and quality requires careful scheduling of resources and capacity allocation [6].

Contemporary cloud application and Edge computing [7] orchestration systems rely on controller/worker design patterns to allocate, distribute, and manage resources [8]. Standard solutions like Cloud Foundry [9], OpenShift [10], Apache Mesos [11], Docker Swarm [12], and Kubernetes [13] span across multiple zones at data centers, multiple global regions, and even telecommunication systems point of presence locations [14]. Current data center design is predicated on the assumption of centralized orchestration to manage and distribute resources [15][8] [16][17][18]. These systems usually operate across multiple zones in the data center or even various regions. Previous research has concluded that random network partitions cannot be avoided in these scenarios [19], leaving system designers to choose between consistency and availability, as defined by the CAP theorem [20].

Controller/worker architectures guarantee configuration consistency via the employment of redundant storage systems, in most cases coordinated via consensus algorithms such as Paxos or Raft. These algorithms ensure information consistency against network failures while decreasing availability as network regions increase [21].

In addition, intrinsic to the centralized architecture design is the requirement to implement strong security measures. It only requires the security compromise of the controller nodes in the system to take control of the entire network. It is common for system designers to isolate controller nodes from the application data plane [2] to restrict orchestrated application access to the control plane, further increasing the deployment complexity across network boundaries.

This research will analyze decentralized architectures for resource management by applying the recent practical advancements in blockchain and consensus technology. The proposed blockchain-based decentralized architecture noticeably increases the system availability, including cases of network partitioning, without a significant impact on configuration consistency.

1.1. Datacenter Management Architecture

Current data center management technologies rely on centralized architectures modeled using the controller/worker pattern. A controller entity receives one or more requests and then communicates with worker entities to execute them.

The controller/worker (Figure 1) pattern allows systems designers to simplify the selection, parameterization, and scheduling of resources by operating under the assumption that a global state view of the system is available [22].

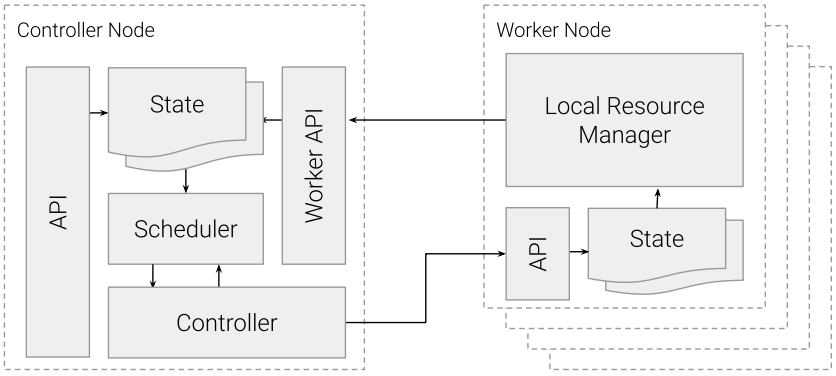

Figure 1 Controller Worker Architecture

In this architecture, the controller and the worker are permanently running a loop to ensure the controller has an up-to-date view of the system and the worker receives the latest scheduled configuration. The ability of a worker node to perform its intended function requires a constant connection with a controller node. If such a link were to be interrupted, the controller could not assume that the node was still executing its last configuration. Conversely, the worker node cannot report or react to application runtime changes, including failures or the average load of the service.

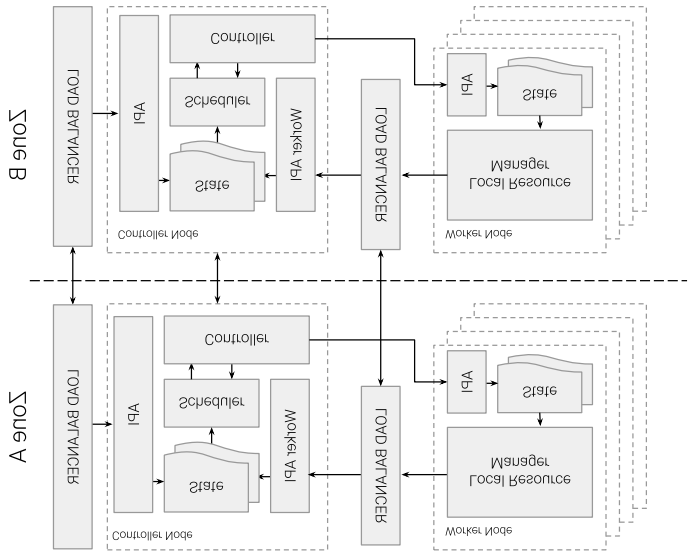

Figure 2 Redundant Controller Worker

As these systems' topology complexity and scale increase, many questions arise, such as latency, reliability, and load balancing [23]. In most cases, the controller as a single point of failure is replicated and strategically placed to minimize the impact of hardware failures. Replicated controllers (Figure 2) can be located across different hardware replicated zones in the data center or across multiple data centers altogether.

System availability against machine failures is typically addressed through controller and worker redundancy. However, this topology introduces the possibility of network partitions, forcing system designers to make compromises between consistency & availability. Per the CAP Theorem [20], any distributed data store can only provide two of three guarantees: Consistency, Availability, and Partition Tolerance (Figure 3).

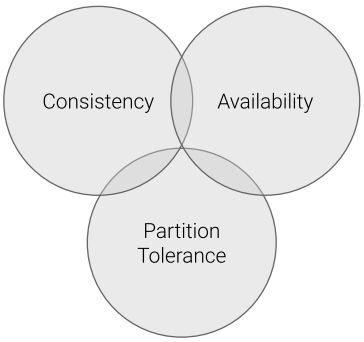

Figure 3 CAP Theorem

- Consistency is the property where the system's state is available to every node after receiving the most recent update.
- Availability is the property in which every request is processed successfully without the guarantee that it contains the most recent update.
- Partition tolerance is the system's ability to operate even when communication between nodes is interrupted continuously.

Because system designers cannot prevent network failures, the compromise between consistency and availability is typically addressed by implementing consensus schemes such as the Paxos [24] and Raft algorithms [25]. Architectures based on these algorithms require that at least most control nodes are available and that worker nodes can connect to one of those nodes to ensure access to the most recent view of the system. If a controller network connection is interrupted, it can no longer perform its designated function.

However, contemporary web applications are typically designed to satisfy the need for scale, availability, and globally distributed access. These applications are resilient against transient failures to ensure availability across fragile global environments. They do not require absolute consistency of the control plane data [26]. For example, web

applications are typically required to run without any interruption. It is impossible to globally coordinate the deployment of all services simultaneously while updating the code of the Web clients. Web applications are typically written to support interactions with multiple versions of the backend services to solve this problem.

Instead of maximizing the consistency of the configuration of the control plane, those applications can benefit from increased availability of the underlying control system to ensure applications can react to local environmental changes, such as failures and load changes, even if the deployment does not reflect the most recent configuration or code. Additionally, such casual consistency needs to be able to disambiguate configuration divergences as it is impossible to predict how far a system might be disconnected from the centralized configuration store. This design would also introduce support scenarios where a local operator can perform a local change that eventually merges with the rest of the system configuration during network disconnection.

1.2. State-of-the-art Management systems

State-of-the-art application management technologies are focused on simplifying automation via intent-based declarative configuration, where state updates are propagated over time in what is known as the “eventual consistency” mechanism [27]. Finally, all nodes will reflect the most current configuration as scheduled by a central controller.

Examples of controller/worker architecture systems include Cloud Foundry, OpenShift, Apache Mesos, Docker Swarm, and Kubernetes. As previously stated, the architecture of these systems prioritizes intent-record consistency while providing substantial availability through controller replication.

Apache Mesos, Docker Swarm, and Kubernetes store configuration state in Etcd [28], a key-value store, using the Raft consensus algorithm to ensure consistency and partition resistance. Essentially, the controller returns the confirmation to the client only when a quorum of nodes acknowledges the request. Reads are linearizable, implying that once a write is completed, all later read operations should return the value of that write or the value of the last write. Alternatively, Cloud Foundry utilizes MySQL, a relational database that relies on the Paxos algorithm. However, in practical terms, the only difference between Paxos and Raft is the leader’s election mechanism [29].

For example, when a user submits an intent request, the desired configuration change is first stored in either Etcd or MySQL. Depending on the system, the transaction is then confirmed to the user, who reasonably expects the request to be distributed and committed. Once the configuration change is saved, the controller can execute the scheduling algorithm and communicate the changes to the affected worker nodes to achieve a consistent global state that matches the user's intentions. These mechanisms, in aggregate, provide intent-record state consistency that guarantees high statistical availability and good network partition resistance.

1.3. The case for decentralizing the control plane

The case for decentralization of the control plane has been discussed using Conflict-free Replicated Data Types [30] and fragmenting large deployments into federations, requiring a distributed database that spans the entire system [31]. While these solutions increase scale and availability, coordination between the infrastructure providers is necessary to maintain the list of deployed locations. In addition, significant performance issues have been documented when using federation [32], resulting in performance degradation from even minor increases to latency (8.7x) and resource contention (12.0x) in comparison to centralized cluster architectures.

Coordination systems based on blockchain technology have been explored to reduce the management cost of maintaining a centralized repository of federated systems [33]. Using Ethereum or similar general-purpose and permission-less blockchains has been deemed too costly and inefficient [34].

Hierarchical control plane structures have also been studied to solve the problem of orchestrating Software-defined networking across vast deployments. Maintaining a global view of the system with solutions like FlowBroker, D-SDN, Tungsten, or Kandoo does not address challenges with information sharing across network partitions [35].

Additionally, federation requires establishing and managing trust relationships between the different clusters [36], requiring an additional level of governance [37][38] and trust initialization setup. To solve this problem, blockchain for IoT and Edge has been preliminarily studied via decision framework studies [39], showing the potential viability of using permissioned and private blockchains as a control plane for highly distributed environments.

The feasibility of implementing blockchain technology control systems has been demonstrated using a multi-tier architecture to record and distribute configurations across multiple control nodes [40]. Existing implementations leverage smart contracts to substitute access control and preserve the sequence of change requests. However, deployment control is still delegated to a traditional controller/worker cluster architecture. While a fully distributed blockchain across all regions can yield similar results concerning global availability, it still depends on the availability of the local cluster controller to ensure all nodes can be operated. Therefore, it does not impact the CAP properties of the system or the security of the control nodes.

Concerning blockchain performance, previous work has determined that the throughput characteristics of three-tier control systems utilizing a general-purpose blockchain as a record store yield good results [41]. This research evolves previous approaches by integrating control and work nodes into a single hybrid component and using Byzantine resistance consensus algorithms to coordinate the blockchain's agreement, termination, and validity. Existing blockchains implementation like Ethereum [42], Cardano [43], Solana[44], Hyperledger [45], or any other general-purpose blockchain with support for smart contracts can be used to manage and execute purpose-built smart contracts containing the logic of configuration disambiguation, scheduling, and access control. However, using existing blockchains will require a network of Oracles capable of performing active functions, including failure detection. In addition, to ensure the same

level of availability, it would require every node running the software to also operate as a general-purpose blockchain node alongside the required Oracles. We decided against this approach due to the runtime, management, and overhead. Although outside the scope of this research, we consider implementing the solution using general-purpose smart contract blockchains worth studying for Web3 applications that rely on both traditional stacks and smart contracts.

CHAPTER 2. SCIENTIFIC APPROACH

2.1. Methodology

For the development of this research, an incremental approach was chosen. Each phase begins by establishing research goals leading to the development of the design that allows the evaluation of the proposed objectives (Figure 4).

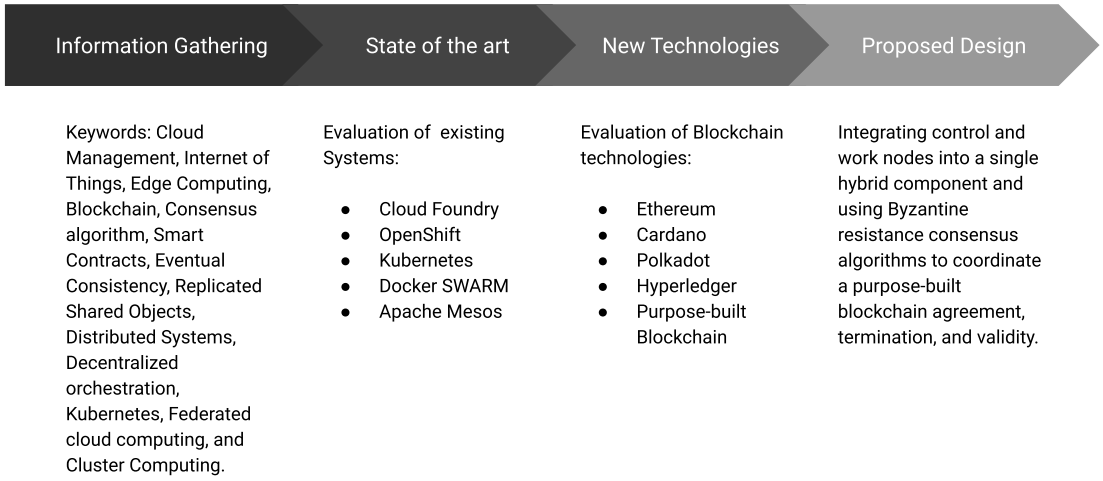

Figure 4 Research Process

The research process was carried out in an initial phase of gathering information and establishing the current condition of the cloud management space, including a thorough evaluation of the most common solutions and present challenges.

The second research phase evaluated the current state of the art regarding decentralization of the control plane solution for cloud systems, software-defined networking solutions, and Internet of Things (IoT) environments. The analysis was not confined to blockchain solutions, yet the benefits showcased by modern blockchain research guided the decisions made for the designed solution.

The third phase determined the solution's requirements, where different integration architectures were evaluated. A final phase in which the proposal is developed, presenting a proposed architecture and analysis of the solution. In the development of each phase, the progress and results obtained were disseminated.

2.2. Information Gathering

Information was collected by searching for articles in various databases, including IEEE Explore, Science Direct, Web of Science, ACM Digital Library, Springer, Google Scholar, and other sources of information. The search keywords were combinations of

the terms: Cloud Management, Internet of Things, Edge Computing, Blockchain, Consensus algorithm, Smart Contracts, Eventual Consistency, Replicated Shared Objects, Distributed Systems, Decentralized orchestration, Kubernetes, Conflict-free replicated datatypes, Federated cloud computing, and Cluster Computing.

In total, 786 documents were found. These were filtered from the keywords and abstracts, thus determining those relevant to the investigation. The research documents used in the development of this work have been referenced.

2.3. Selection of tools, development, and elaboration of the proposal

After completing the analysis of multiple environments, Kubernetes was selected as the candidate to derive from towards implementing the goals of this research for the following reasons:

1. Intent-based configuration model based on resources and stored in a machine-friendly format.
2. The loosely coupled dependency resource model makes it possible to simplify the scheduling of resources through eventual consistency.
3. Flexible architecture enabling direct communication with worker nodes replaces the controller's role with a distributed coordinator.
4. Most adopted solutions with broad support for cloud application management patterns.
5. Built-in support for most cloud platform APIs enables easy deployment on existing cloud environments.

The selection of the decentralization mechanism to integrate control and work nodes into a single hybrid component and use consensus algorithms to implement coordination. In blockchain-centric systems, a natural pattern is decentralizing control and replacing authority with Byzantine-resistant consensus patterns. Applying this pattern to the cloud management space may seem unintuitive at first glance, yet this solution addresses the primary goals of this research. The utilization of blockchain technology at the core of the control plane provides the following benefits:

1. Data is replicated across all nodes by default. Additional optimizations are possible by implementing lightweight nodes that utilize Merklized Abstract Syntax Tree (MAST) and Merkle proofs to validate configuration changes distributed through the system [46].

2. Nodes are self-governing entities that operate when isolated from the cluster in cases of a network partition. Blockchain exhibits high availability [47] properties when assuming causal consistency of the system configuration [48].
3. The diversity of consensus algorithms that enable system designers to choose and balance the properties of the system should be prioritized, including trust setup, security, and incentives [49].
4. Immutable configuration history [50]. Every change in the system is sequentially stored and cryptographically secured in the blockchain, enabling system operators to determine the system's state at any point in time.

CHAPTER 3. RESEARCH OBJECTIVES

3.1 Research Objectives

The objective of this research (Figure 5) is to evaluate the implementation of a highly available and partition-resistance cloud management system using blockchain technology. This research also offers a vision for a solution that addresses the critical challenge by leveraging the capabilities following key technologies:

- Blockchain: A distributed store that can efficiently record transactions between two parties in a verifiable and permanent way [51].
- Nakamoto Consensus: a set of rules that verifies the authenticity of a blockchain network. Also considered the solution to the Byzantine Generals Problem [52].

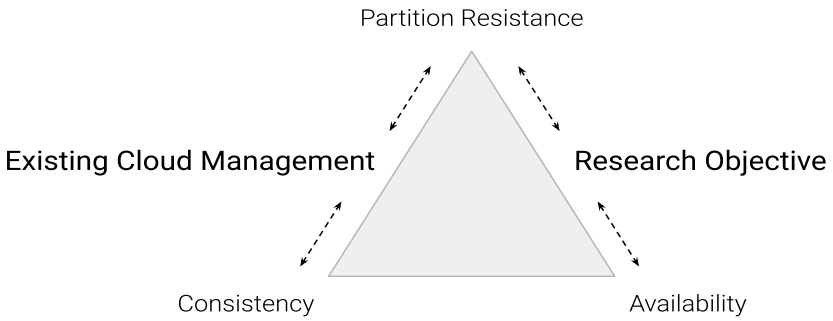

Figure 5 Research Objectives

To date, little practical research has been performed to weaken the criteria for replica consistency to improve the partition tolerance, availability, and performance of cloud systems owing to the non-monotonic nature of the system configuration. Non-monotonicity occurs when a new configuration value invalidates the previous configuration state [48]. However, because of the characteristics of the eventual system consistency described previously, we believe that a system of rules that disambiguates potentially conflicting configuration requests can provide acceptable levels of delayed consistency. For the most part, System operators prioritize their focus on the system's final state and, in most cases, can infer the consequences of intermediate states during configuration changes. Capturing transaction ordering rules into a cryptographic protocol and persisting results on a blockchain presents an opportunity to leverage mainstream consensus algorithms to solve the challenges presented by the CAP theorem.

Combining these technologies has made it possible to design Peer to Peer systems focusing on maximum availability while providing a verifiable recorded history of system state changes. In a world where distributed systems are becoming increasingly heterogeneous, decentralization could be construed as the solution for heterogeneous scalability.

3.2. Dissemination of results

We have used the following criteria to select the most suitable journal:

1. The theme of the magazine.
2. Impact factor and quartile in all its categories related to our area of knowledge.

The impact factor report of the article used to present the doctoral work:

Title: Blockchain-based Cloud Management Architecture for Maximum Availability.

Research-specific objective accomplished: Analyze the impact on the availability of blockchain-based decentralized systems and compare them with the current centralized approach.

Authors: Alberto Arias Maestro, Oscar Sanjuan Martinez, Ankur M. Teredesai and Vicente García-Díaz

Journal: The International Journal of Interactive Multimedia and Artificial Intelligence

Impact factor 4.936 (JCR 2021)

3.3. Research Impact

I have been fortunate to be part of the Cloud industry for the last 15 years. From the emergence of the software-defined data center, virtualization of computing and network resources, multi-cloud orchestration, containers, and back hybrid cloud computing. The common theme across all of these technologies is the default implementation of centralized orchestration to manage and distribute resources to enable a wide range of scale requirements.

Table 1 shows the impact of the relevant research published and patented at the time of this writing.

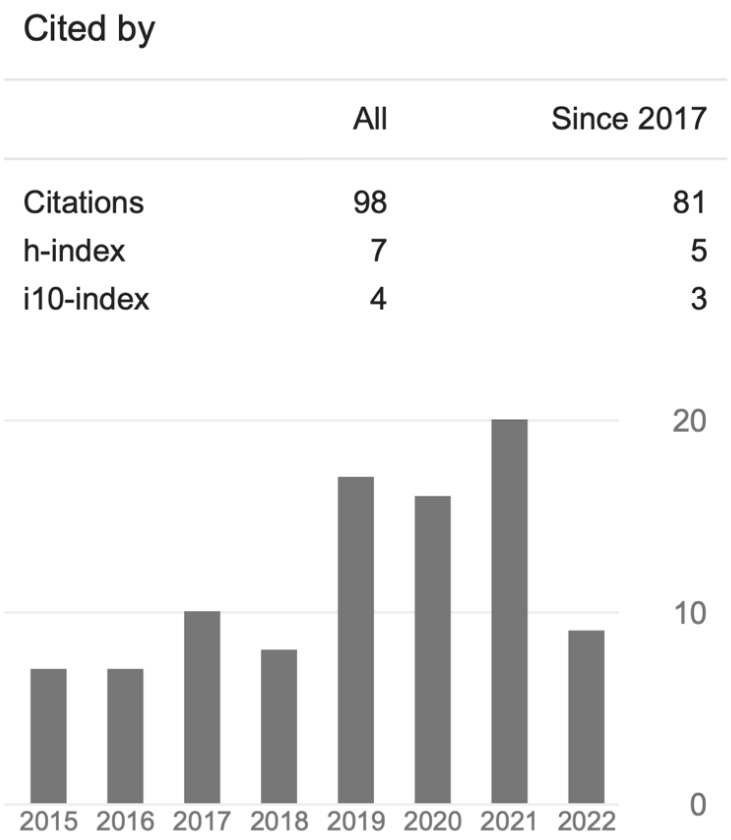

Table 1 Impact Summary

3.4. Patents

Title	Cited	Year
Adaptive autoscaling for virtualized applications. US Patent 9,817,699	31	2017
Computer relational database method and system having role-based access control. US Patent 10,430,430 US Patent 9,058,353 US Patent 9,852,206	26	2019
Asynchronous programming model for concurrent	8	2014

workflow scenarios.		
Distributed event system for relational models. US Patent 9,384,361 US Patent 9,195,707	14	2016
Workflow design for long-running distributed operations using no SQL databases. US Patent App. 14/206,342	4	2014
Interface infrastructure for a continuation-based runtime. US Patent 9,916,136 US Patent 9,354,847	3	2018

PART II. DESIGN PROPOSAL

CHAPTER 4. CLOUD SERVICES

This proposal is structured into four sections. First, a Kubernetes overview is provided as the foundational orchestration system to be evolved towards decentralization. The second section covers the architecture of the hybrid controller/worker node and its connectivity to other nodes. The third section describes how the system state configuration is encoded into the blockchain structure, followed by global ordering rules that ensure transaction validity to be applied by participating nodes.

4.1. Kubernetes concepts and architecture

The best-known example of this architecture is Kubernetes. In Kubernetes, the system architecture prioritizes intent-record consistency while providing substantial availability through the replication of the controller [31].

Kubernetes stores configuration states in Etcd, a key-value store using the Raft consensus algorithm to ensure consistency and partition resistance. In essence, only when a quorum of nodes confirms the write the node returns a confirmation to the client.

To preserve configuration consistency, operations are linearizable. Once a write completes, all later read operations should return the value of that write or the value of a subsequent write. Configuration history linearizability ensures that all operations that modify the configuration are sequential and have already considered any last changes made [53].

Like in most modern orchestration systems, Kubernetes utilizes a declarative configuration model to reduce the possibility of commutative order violations where the order of the configuration is reversed. Declarative configuration models aggregate applications' deployment configuration in one or several documents that represent the final state of the system instead of individual API calls to manipulate the state of the system.

In Kubernetes, when a user submits a change request, the desired intent configuration is first communicated as a document containing resource definitions. Resource definitions are declarations of required resources, application binaries, network configurations, and deployment configurations (Figure 6).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
```

```
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Figure 6 Example Nginx deployment

The controller stores the deployment configurations and processes the requests determining the best allocation of resources across the system to reasonably satisfy the demand. The final deployment configuration is then communicated to the worker nodes, converging towards a consistent state that matches the user's intentions.

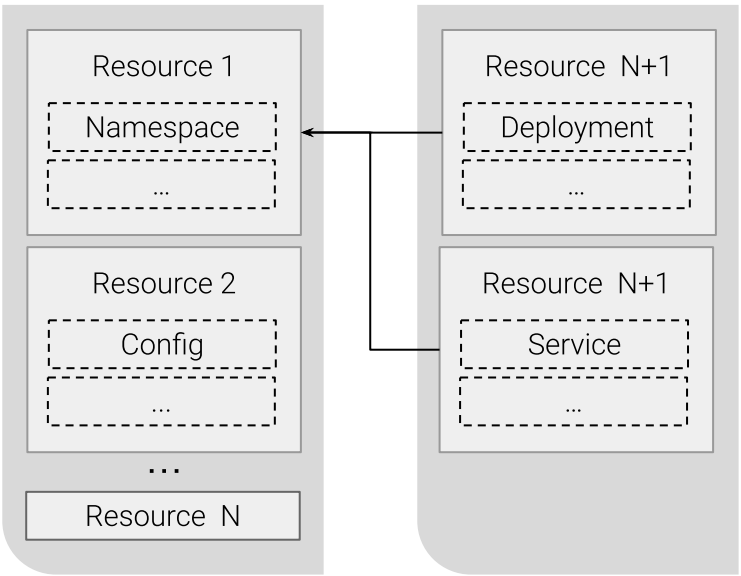

Figure 7 Linearizable configuration example

Configuration dependencies are decoupled as much as possible to maximize the ability to change the system's configuration over time. There are two types of dependencies: containment and association.

A containment relationship requires the contained resource to be created after the container object (Figure 7). For example, deployment must be contained within a namespace, and therefore, the creation order cannot be reversed. The consistency provided through the underlying ETCD instance ensures that the system does not have to make disambiguation decisions when two controller nodes try to update conflicting configuration changes.

An association dependency is an indirect reference to another resource through labels defined during resource creation. This mechanism allows two resources to maintain a separate creation and maintenance lifecycle. For example, a Service resource that exposes a container endpoint can be created before the container itself. This mechanism enables deployment patterns like blue-green and canary deployments [54]. In canary implementations (Figure 8), more than one application instance is deployed while the network traffic load is balanced. Canary deployments enable the gradual deployment and rollback of application versions.

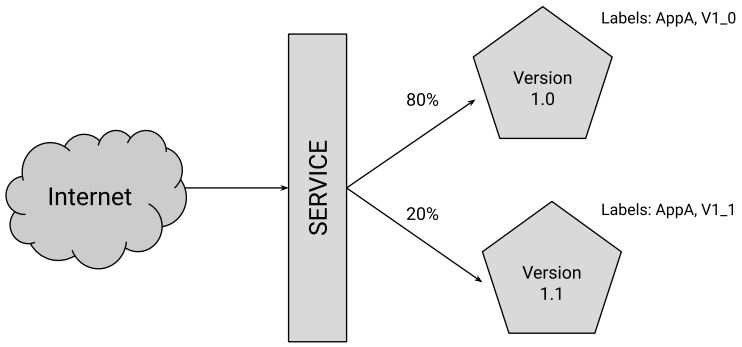

Figure 8 Canary Deployment

Users can determine the system's state by requesting an up-to-date view of previously submitted resources. All resources have spec and status fields (Figure 9). The spec is the original resource specification where users declare the desired state. The status field can only be modified by the system and contains the actual known state of the object by the controller.

```
kind: Pod
...
spec:
  readinessGates:
    - conditionType: "www.deployment.com/check"
status:
```

```
  conditions:
    - type: Ready
      status: "False"
      lastProbeTime: null
      lastTransitionTime: 2022-01-01T00:00:00Z
    - type: "www.deployment.com/ check”
      status: "False"
      lastProbeTime: null
      lastTransitionTime: 2022-01-01T00:00:00Z
  containerStatuses:
    - containerID: 88888888
      ready: true
...
```

Figure 9 Pod status

The result of this architecture is that Kubernetes provides intent-record state consistency guarantees with high statistical availability. User intent consistency is guaranteed by the underlying storage system (ETCD). However, while in most cases, the end state of the system is also consistent, the controller can only provide the last know state of the system or report the last time the status was communicated to the controller. If a Kubernetes node fails, the controller node will restart pods, detach the volumes, wait for the old volumes to detach, and reuse the volumes across a new node. Typically, these steps would take about 5 to 10 minutes.

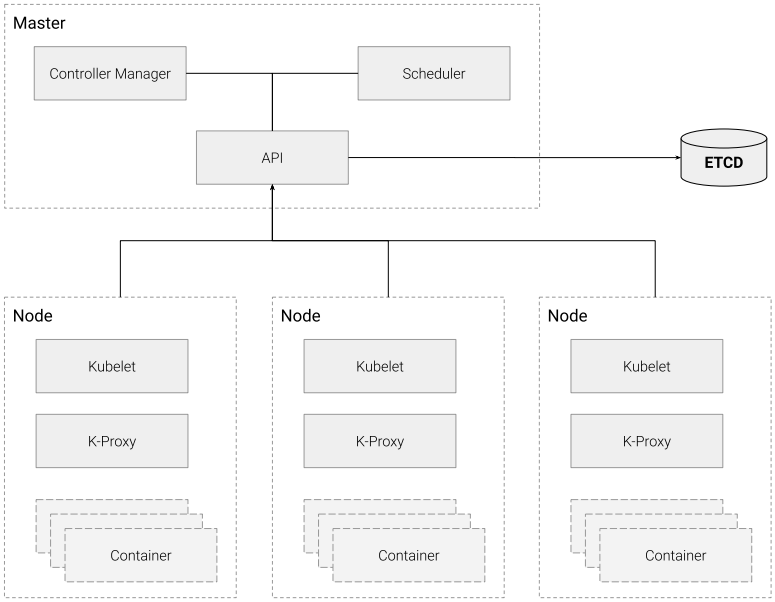

Figure 10 Kubernetes Architecture

To understand how the system state is represented and is essential to understand the following components (Figure 10) that will be further referenced across this research:

- Cluster: A collection of nodes used by Kubernetes to run application workloads orchestrated by one or more replicated controller nodes.
- Nodes: A physical or virtual machine that provides computing, memory, storage, and networking resources to run applications.
- Master: A node responsible for storing the global state of the system, scheduling resources, handling events, and enforcing security policies.
- Pods: Represent a group of containers that are scheduled and deployed together. Applications are typically composed of several pods.
- Labels: Key-value pairs that are used to reference and group managed resources. Labels are used to identify and represent the relationship between resources uniquely.
- Label selector: Used to select resources across the system.
- Annotations: Used to attach arbitrary metadata to resources.
- Services: Used to publish and expose application network endpoints running on o a set of Pods determined by a selector.
- Volume: Ephemeral storage accessible from within a pod.
- ReplicationControllers: Manage a group of pods identified by a selector, ensuring that the intended number of pods is running.
- ReplicaSet: This performs the same function as a replication controller but with additional capabilities to select the set of pods to manage.
- StateFullSet: Like replication controllers with the distinction of specifying which nodes run which pods enable better control for state-full workloads.
- Secrets: Store credentials or other sensitive information.
- Namespace: A virtual partition of resources enabling the separation of resources across clusters.

Kubernetes clusters have at least one controller node and as many worker nodes as necessary to host the intended applications required capacity (Figure 10). While it is possible to separate the different components of the control plane across multiple servers, most controller nodes host the following processes:

- API server: Exposes a REST API used to interact with the controller.
- ETCD: Distributed data store used to persist the state and configuration of the cluster.
- Controller manager: Used to manage and configure external services used by the cluster. Including load balancers, persistent volumes, name servers, etc.
- Scheduler: Responsible for scheduling resources into nodes following the policies specified in the cluster configuration.

Kubernetes worker nodes execute control place processes alongside user applications using containers to provide the necessary security isolation. Worker nodes require a direct connection to the controller node. The control plane network might be segregated using separate physical or virtual networking interfaces. Worker nodes have the following processes:

- Kubelet: Communicates with controller and configures the node to run pods as scheduled by the controller node. Includes running containers, mounting nodes, reporting the node's state to the controller, etc.
- Proxy: Manages the node's network configuration to meet the configured services' requirements.

While Kubernetes has emerged as the primary choice for containerized applications, large deployments suffer from latency and availability issues reducing its suitability for highly distributed environments [55]. In addition, Kubernetes is highly sensitive to network partitioning, resulting in the reduced availability of applications running within the cluster [56]. In-depth studies towards providing models to increase the availability of applications running in Kubernetes clusters conclude that service outages can be significantly higher than expected under specific configurations [57]. Solutions have been evaluated to reduce the recovery time of applications within the cluster when a node fails [58]. However, those solutions do not address the situation when the control plane fails, or network connection interruptions make it impossible for a controller to communicate with worker nodes. This situation cannot be mitigated if the control and work functions are independent while requiring active communication.

CHAPTER 5. BYZANTINE RESISTANT ARCHITECTURE

This research evolves previous approaches by integrating control and work nodes into a single hybrid component and using Byzantine resistance consensus algorithms to coordinate the blockchain's agreement, termination, and validity. This yields a peer-to-peer architecture of compute nodes collectively converging into a state that matches the sequence of intents stored in the blockchain. While the primary function of nodes is to host workloads, nodes maintain a full copy of the blockchain and participate in the consensus process both as block creators and validators.

For this research, we will incorporate elements of the existing Kubernetes architecture. In particular, the Kubelet component provides the actuation of state configuration changes by communicating with the node operating system. In essence, workloads are hosted by deploying containers in Pods. Essential to the spirit of most cloud management systems, a node may be a virtual or physical machine, depending on the cluster.

5.1. Node to node communication

Nodes connect to other nodes using a P2P Gossip protocol (Figure 11). The initial discovery is made through dynamic DNS. Once a node can connect to other nodes, it will be able to receive the list of known nodes and blocks.

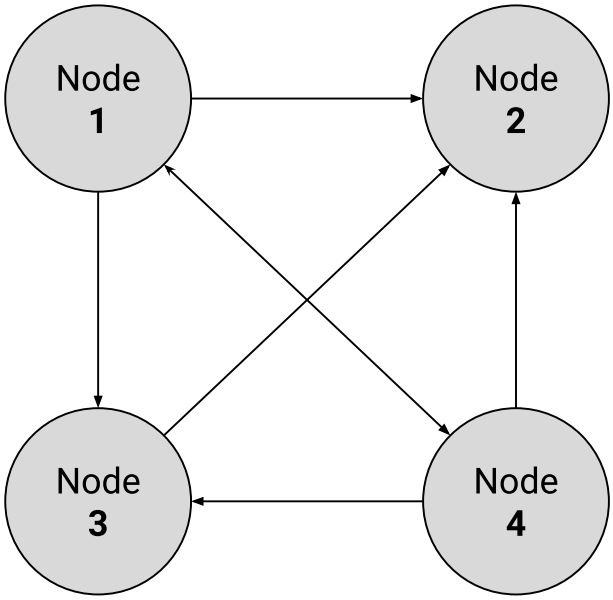

Figure 11 Gossip Protocol

The number of nodes in the cluster might change during the life of the cluster. This mechanism allows nodes to discover the topology of the network without the need for a central catalog. To accelerate the discovery and prevent Eclipse attacks [59], nodes might be initialized with a list of permanent nodes maintained by the system administrator. Eclipse attacks occur when the first connection of a node to another node is to a compromised node that steers the node to a compromised chain (Figure 12).

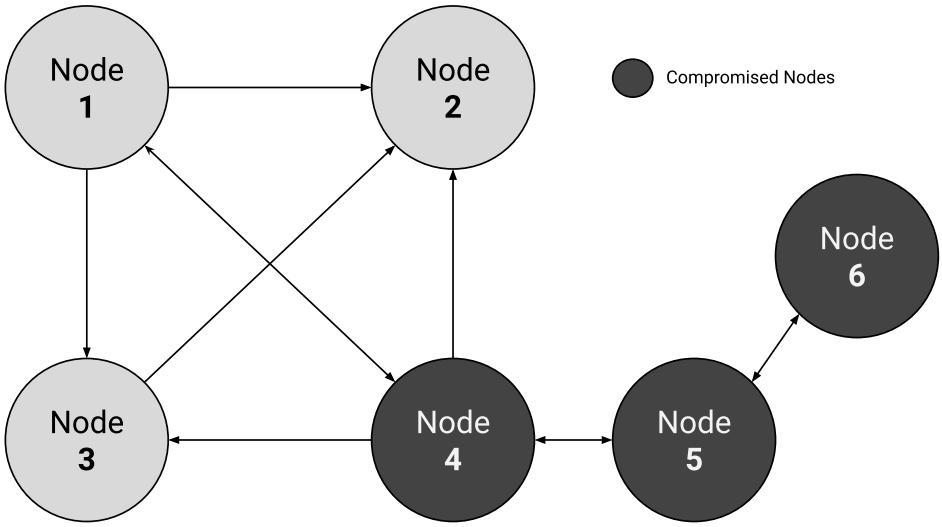

Figure 12 Eclipse Attack

When a node receives a message that a new node has been added to the network it appends it to a list of known nodes. At this point, the new node has not been verified. To prevent nodes from consuming too many resources to maintain communication with the rest of the network, the node limits the number of active connections to the rest of the network. For example, in a cluster with 100 nodes limiting the number of connections to 5 will result in an acceptable delay in the communication across the network [60].

An additional security procedure includes selecting different nodes to connect to when the node is restarted, preventing further crashes if the reason for the restart is that another node exploited a bug that made the node crash. To minimize the risk of compromise, when a node processes the list of known nodes it uses feeler connections to detect peer node anomalies [61].

Finally, to prevent compromised nodes from flooding other nodes with randomized addresses, the number of addresses that a node can receive is rate limited [62]. In addition, working nodes would only broadcast addresses that have already been verified, therefore truncating the range of such attack.

Nodes are also registered in the blockchains, if a node obtains a full copy of the blockchain it will be able to validate the list of nodes received with the configuration

stored in the blockchain. There might be additional security guarantees when adding a node to the network, depending on the consensus algorithm, like, certificate chain validation in Proof of Authority. Notice that it is possible to have nodes as part of the network but don't participate in the consensus algorithm yet communicate with other nodes and maintain a full copy of the blockchain.

Nodes can also receive direct connections from users. Users submit new transactions and inspect the state of the node and the last known state of the chain by that node. Nodes connect to nodes via an RPC API, enabling them to interact with the cluster without the need to store a copy of the blockchain and participate in the P2P network.

5.2. Node Architecure

Unlike in Kubernetes masters, nodes are assigned the responsibility to communicate with external services, for example, updating a DNS entry or configuring a new load balancer. As such, the components of a hybrid node include:

1) The peer manager is responsible for maintaining a list of known peers. It creates and maintains TCP connections and receives new network connections from other peers. The peer manager is responsible for communicating with other nodes via the P2P gossip protocol.

2) The consensus manager is dedicated to applying consensus rules to maintain the longest valid chain known by the node by determining which blocks should be added to the chain or even discarding dead-end chains. The consensus manager is integrated very closely with the peer manager, such that it can adapt the node chain to new information, including blocks and alternative chains. In addition, a node, depending on the consensus algorithm, may be selected for mining a new block. The consensus manager is responsible for communicating the new block to the other peer nodes.

3) The validator is responsible for analyzing the contents of a block and ensuring that all new transactions are valid. Transaction order, transaction inputs and outputs, locking script execution, and any other block rules are related to the consensus algorithm.

4) Pending changes comprise a list of known pending transactions. Each block maintains a list of the pending transactions. When a new block is received or minted, the transactions in the block are removed from the pending list.

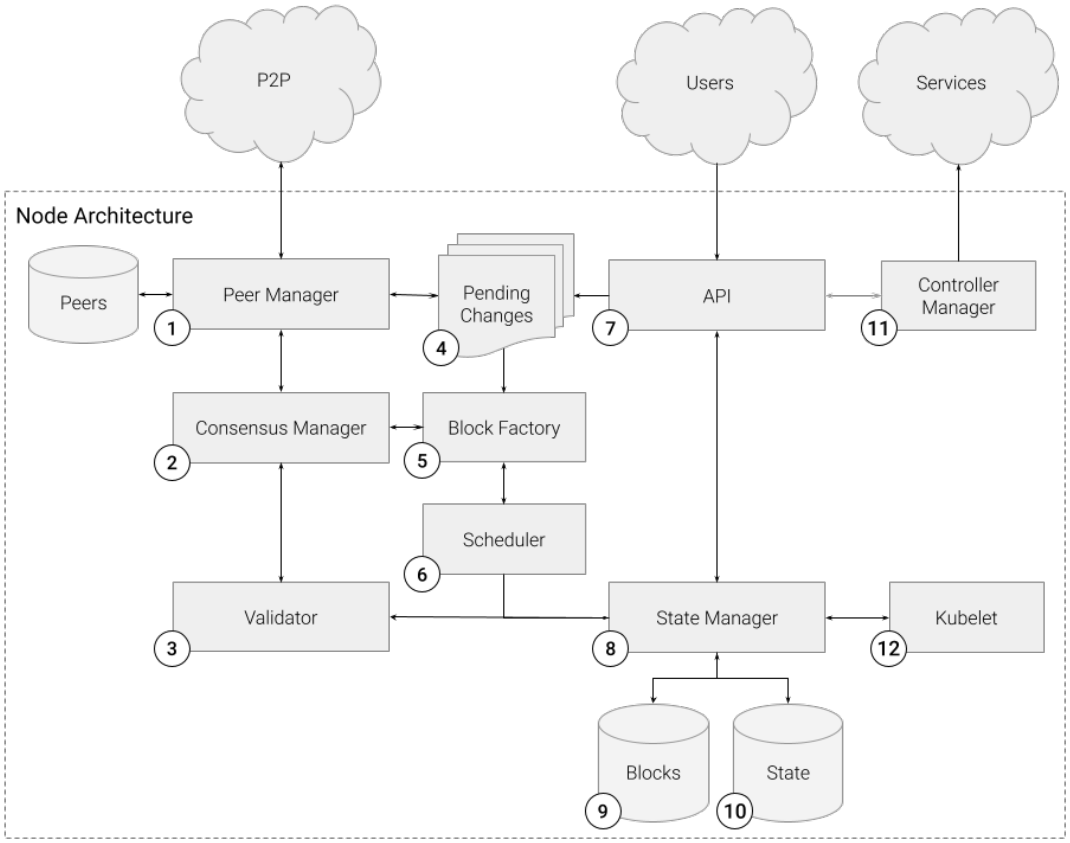

Figure 13 Hybrid Node architecture

5) The block factory is responsible for mining a new block based on the inputs of the pending change list. It communicates with the consensus manager, ensuring that the block is valid by verifying with the validator. Any invalid transactions are reported until a block is correct and ready to be communicated.

6) The scheduler is a component that watches for newly created resources with no assigned nodes.

7) API is the front end of the contents of the state of the cluster and transaction management. Users connect to the node via an API to interact with the cluster without directly operating a node.

8) State Manager maintains the databases and indexes required to store and operate the cluster.

9) Blocks are key-value pair databases indexing every block and transaction of the blockchain by its hash value.

10) State is a document-oriented database with content resulting from executing all transactions in the blockchain.

11) The controller manager is responsible for maintaining the configuration and state of the services external to the cluster.

12) Kubelet is part of the Kubernetes architecture. It is responsible for connecting to the Docker runtime and ensuring that all pods and containers run according to the cluster state determined by the blockchain.

5.3. Blockchain structure

Transaction data is stored in blocks organized into a linear sequence over time. New transactions are added to blocks, and blocks are added at the blockchain's end (Figure 14). Unlike traditional cryptocurrency blockchains, the structure of this chain does not keep track of a ledger, instead, processing the content of the blockchain results in a hierarchal tree of resources that represent the system state.

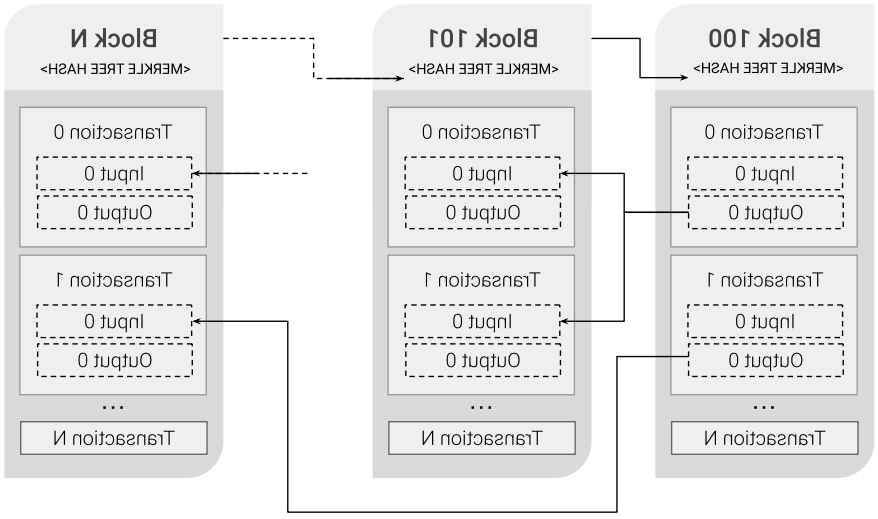

Figure 14 Blockchain structure

Each block contains the hash of each transaction calculated by adding every transaction into a Merkle tree and storing the root as part of the block header. This mechanism allows rapid verification that a transaction is part of the block (Figure 15). For a user to verify whether Transaction 3 is in the blockchain, it only needs to compute the hashes of the Merkle tree and check the Root Hash of the block where the transaction is stored. Notice that the user does not need to calculate the hash of all the transactions in the block, only the hash of the sibling transaction within the tree.

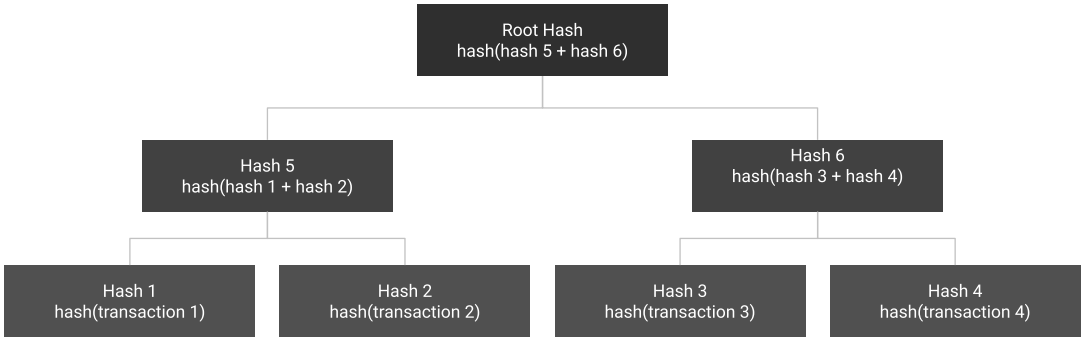

Figure 15 Transaction Merkle Tree

In addition, every block includes the hash of the previous block's header. In essence, every time a block is added to the chain, the more complicated and more challenging it is to change or remove previous blocks as it would require calculating new hashes for every block after the one being modified. In practical terms, every transaction in the blockchain is irreversible and final.

Even in the extreme case that the whole blockchain gets updated to masquerade a change, cluster operators only need to store the Merkle root hash of blocks periodically to be able to identify whether the blockchain has been tampered with.

Because of the verification properties of Merkle trees, it is possible to implement lightweight nodes that do not perform all the functions of a full node. A lightweight node might delete the blocks of the chain (keeping the Merkle trees) after processing them and calculating the current configuration resource tree. When a new block arrives, it verifies the block and processes it, the node can still verify whether the transactions part of the new block was part of the original chain by just checking the Merkle tree.

It is possible to implement further optimizations where a lightweight node only receives transactions that affect the status of the node. The lightweight node can still verify that does' transactions belong to a block of the chain and update part of the resource tree.

Future research will analyze the possibility of using this mechanism to create large federations of clusters with full nodes that orchestrate cross federation data and local nodes that only store the part of the chain that contains resource information relevant to the federation.

5.5. Block Structure

Blocks are structured using the following format:

Field		Description	Bytes
Magic Number		Unique fixed value used to identify the start of a new block. The value is always: 0x15042011	4
Size		Size of the block	4
Header	Version	Block version	4
	Hash Previous Block	SHA256 of the previous block header	32
	Hash Merkle Root	SHA256 of the Merkle root of the Merkle tree of all the transactions in the block	32
	Time	Block timestamp	4
Transaction Counter		Positive integer with the number of transactions in the block	4
Transactions		List of transactions	Variable
Difficulty Target*		The proof-of-work algorithm difficulty target for this block	4
Nonce*		A counter used for the proof-of-work algorithm	4

*Difficulty Target and Nonce are only used when proof of work is used as a consensus algorithm.

Transactions are stored within blocks using the following structure:

Field		Description	Bytes
Number of Inputs		Positive integer with the number of inputs in the transactions	2
Inputs - For each Input	Hash Previous Transaction	SHA256 of previous transaction	Variable
	Previous Output Index	Index of the output of the previous transaction	2
	Script Length	Length of the script	4
	Script	Script Contents	Variable
sequence_no		normally 0xFFFFFFFF; irrelevant unless transaction's lock_time is > 0	4
Number of Outputs		Positive integer with the number of outputs in the transactions	2
List of outputs			
Outputs - For each output	Value Length	Length of the value	4
	Value	Transaction’s value content	Variable
	Script Length	Length of the script	4
	Script	Script Contents	Variable
lock_time		if non-zero and sequence numbers are < 0xFFFFFFFF: block height or timestamp when transaction is final	4

5.6. Network Partitioning

Network partitioning occurs when a group of nodes is isolated and cannot communicate with the remaining nodes in the network. This is a common scenario when those nodes are not in the same data center, or the data center is partitioned into two or more availability zones. Note that in the proposed architecture, when a network partition occurs, there is a risk that transactions submitted to the partition with the shortest chain will become invalid once the network connectivity is restored. The transactions are appended to the Pending Changes list (Figure 16).

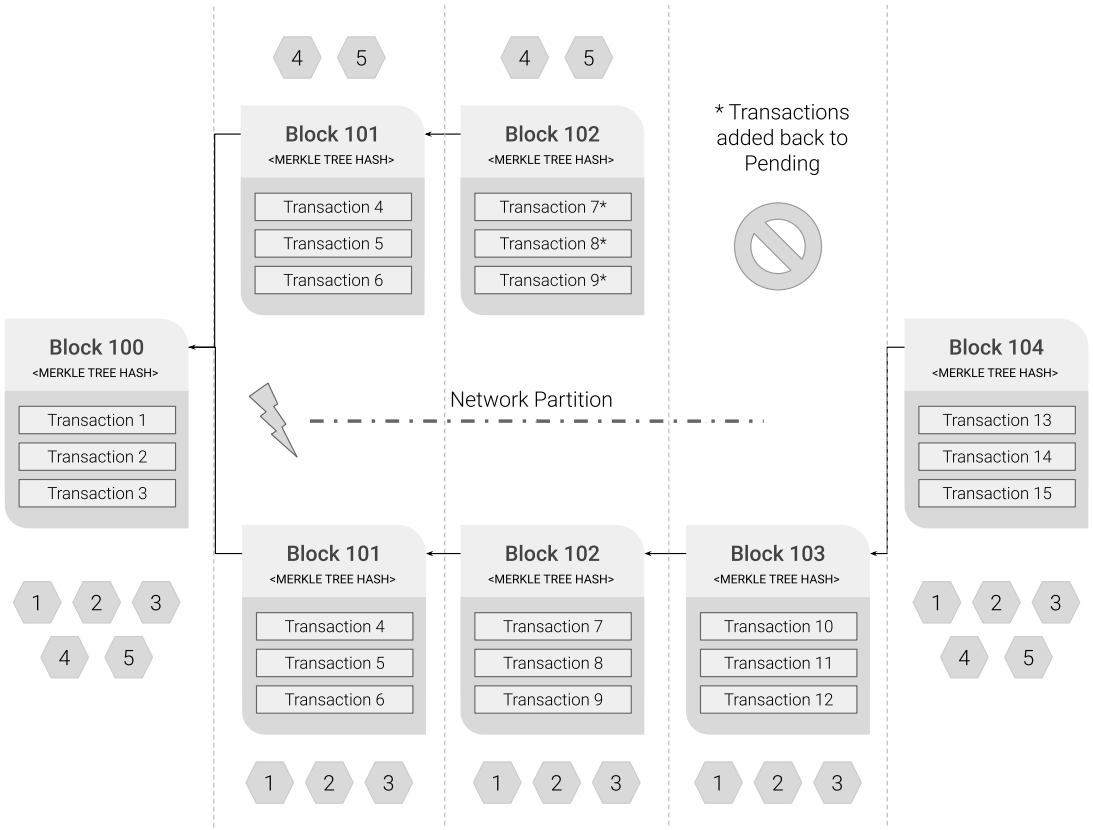

Figure 16 Chain resolution after Network Partition

So far, we have discussed the core components and behaviors of the system. From the analysis conducted throughout this research, we can deduce that the system meets the following propositions:

Proposition 1: Any node can accept a transaction.

Proposition 2: A single node can add a block to the chain.

Proposition 3: Nodes do not require connection to other nodes to accept transactions.

Proposition 4: A group of nodes (more than one node), where each node can connect to others, will generate a chain faster than a group with fewer nodes.

Proposition 5: A node will always accept the longest chain available.

Therefore, we can formulate the following three theorems by mathematical logic:

Theorem: Maximum Availability: If a node is available, the system is available.

Proof of Theorem 1. P1 ∨ P2 ∨ P3 ⟹ T1. If any node can accept a transaction (Proposition 1), and a single node can add a block to the chain (Proposition 2), and nodes do not require the connection to other nodes to accept transactions (Proposition 3), then if a node is available, the system is available.

Theorem: Eventual Consistency: A transaction can only be considered irreversibly committed when it is part of a block that is in the longest chain and is part of the current chain for most of the nodes in the network.

Proof of Theorem 2. P3 ∨ P4 ⟹T2. If nodes do not require the connection to other nodes to accept transactions (Proposition 3), and a group of nodes (more than one node), where each node can connect to others, will generate a chain faster than a group with fewer nodes (Proposition 4), then a transaction can only be considered irreversibly committed when it is part of a block that is in the longest chain, and it is part of the current chain for most of the nodes in the network.

Theorem: Partition Primacy: A network partition with the majority of nodes generates the longest chain with irreversibly committed transactions.

Proof of Theorem 3. P4 ∨ P5 ⟹T3. If a group of nodes (more than one node), where each node can connect to others, will generate a chain faster than a group with fewer nodes (Proposition 4), and a node will always accept the longest chain available (Proposition 5), then a network partition with the majority of nodes generates the longest chain with irreversibly committed transactions.

5.7. Availability Examples

Traditional Paxos/Raft-based systems are available if most replica nodes are available to achieve quorum and maintain the configuration store consistency (Table 2). When there are three zones, both systems are reliable when one fault occurs. However, the differences are revealed when two Paxos/Raft replicas fail, preventing the system from

achieving a quorum and leading to system failure. Note that in this proposal (Table 3), only users who can access a partition with available nodes will be able to submit transactions.

Zones/Replicas	Replica Faults	Partitions	Paxos/Raft
3 / 3	1	0	Available
3 / 3	2	0	Fault
3 / 3	0	2	Fault
9 / 9	4	0	Available
9 / 9	5	0	Fault
9 / 9	0	3	Fault
3 / 3	1	0	Available
3 / 3	2	0	Fault

Table 2 Availability examples of Paxos/Raft

Additionally, as stated in the Partition Primacy and Eventual Consistency theorems, only nodes in the largest partition will be able to confirm transactions irreversibly.

Zones/Replicas	Replica Faults	Partitions	Proposed
3 / 300	50 / 50 / 50	0	Available
3 / 300	100 / 0 / 0	0	Available[1]
3 / 300	100 / 100 / 100	0	Fault
3 / 300	100 / 0 / 0	1	Available[1]
3 / 300	50 / 50 / 50	1	Available[2]
3 / 300	50 / 50 / 50	2	Available[2]
3 / 300	50 / 50 / 50	3	Available[2]

[1] Not accessible from failed partitions.

[2] Transactions cannot be considered irreversible until restored

Table 3 Availability examples of the proposed solution

In the Paxos/Raft system, when the number of zones is expanded to nine, and thus, the number of replicas, the statistical availability increases dramatically. However, in cases where multiple network partitions occur, the system can become unavailable because of the inability of replicas to talk to each other and thus prevent a quorum, even with no replica failures. As stated in the theorem of maximum availability, our proposal becomes unavailable only when all the nodes fail.

CHAPTER 6. CLUSTER MANAGEMENT

6.1. Transaction Script language

A script is a list of instructions recorded with each transaction that describes how the next transaction can modify the resource's state specified in the output. Every transaction except the first one in the blockchain consumes the output of a previous transaction (Figure 17).

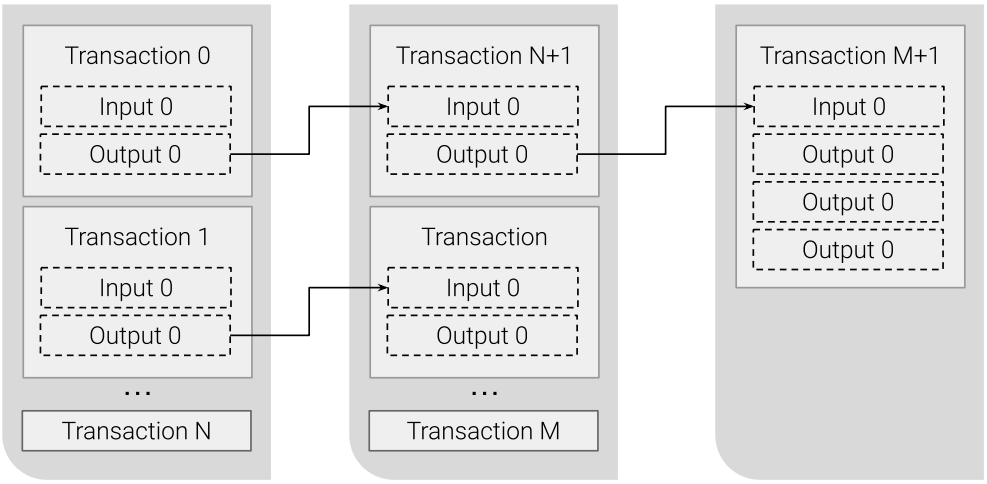

Figure 17 Transaction input-output

Access restrictions are cryptographically secure. When a user submits a transaction to operate an existing resource or to create a new resource within the cluster, the user must satisfy the conditions specified by the input transactions referenced by the new transaction. For example, a new deployment references a namespace where the application resources will be created. The most common restriction is to provide proof of ownership of the private key used to create the previous resource by providing the following:

- The public key matches the hash given in the script of the redeemed transaction output.

- An ECDSA signature over a hash of a simplified version of the transaction. Combined with the public key, it proves authorized users created the transaction.

Scripting provides the flexibility to change the parameters of what's needed to operate resources. For example, the scripting system could be used to require two private keys, a combination of several keys, or even no keys at all.

The script is a Forth-like [63], stack-based, reverse-polish, Turing incomplete language. Reverse polish notation is a system where the operators follow the operands. Operators include memory manipulation, math, loops, function calls, and everything you find in procedural programming languages. Some key facts about scripts:

- Scripts do not have any loop. There are no opcodes that allow jumping to a previous part of the script.
- Scripts always terminate. There are no opcodes that allow the script cannot be halted or paused for any reason.
- Script memory access is stack-based. All input information available to the script is placed in the stack before execution.

Because of these properties, the amount of computation required to execute a script is proportional to its size. This ensures that nodes cannot be stuck running a script indefinitely, an undecidable problem attributed to Turing-complete machines [64].

Scripts are typically presented in a human-readable format where instead of using hexadecimal representation, operand codes are shown as strings prefixed by “OP_”, and data arguments are presented in alphanumerical format. The processing of these scripts is sequential and all data is stored in the stack, there are no variables.

For example, the following opcodes push the numbers 1 to 3 onto the stack:

```
OP_1
OP_2
OP_3
```

Which, when stored in a transaction, will look like 0x515253. Where OP_1 hexadecimal is 0x51, OP_2 is 0x52 and OP_3 is 0x_53.

Several opcodes are provided to push custom data. The previous example can also be written as push 1 bytes using the OP_PUSHDATA1:

```
OP_PUSHDATA1 1 1
OP_PUSHDATA1 1 2
OP_PUSHDATA1 1 3
```

Which, when stored in a transaction, will look like 0x 0x04C010104C010204C0103. The OP_PUSHDATA1 operand is represented as 0x4C followed by 1 byte for each argument.

While scripts can be represented in multiple forms, however, scripts are not malleable. In other words, scripts must be preserved as submitted to the node as part of the transaction to ensure the signature does not change. When computing the hash of the transaction, strict format rules must be implemented to ensure that changes in the signature format, which still makes the content of the transaction valid, do not change the transaction's hash. If this measure is not implemented, then an attacker can disrupt the cluster's operations by creating alternative versions of the same system operation, making it difficult for the user to understand why their original transaction is failing. The sequence of the events would be as follows:

1. User submits transaction A
2. Malicious node takes version A and modifies the format of the signature, therefore creating a new transaction B
3. Both transactions are submitted to nodes to be added to the next block. Both transactions cannot be added because A would invalidate B or vice versa.
4. If transaction A is added to the block, the attack failed
5. If transaction B is added to the block, A becomes invalid.
6. The user might assume A was added to the block, and any future transaction referencing A will fail.
7. The user needs to research and find the existence of transaction B

6.2. Script language Opcodes

Opcodes can be categorized by types of operations like constant declaration, stack manipulation, control flow, encryption, etc.

The following opcodes are used to push constant data onto the stack:

Opcode	Hex	Description
OP_DATA_1 - OP_DATA_75	0x01-0x4b	The next 1-75 bytes is data to be pushed onto the stack.
OP_PUSHDATA1	0x4c	The next byte contains the number of bytes to be pushed onto the stack.
OP_PUSHDATA2	0x4d	The following two bytes contain the number of bytes to be pushed onto the stack in little-endian order.
OP_PUSHDATA4	0x4e	The next four bytes contain the number of bytes to be pushed onto the stack in little-endian order.
OP_1NEGATE	0x4f	The number -1 is pushed onto the stack.
OP_FALSE/OP_0	0x00	The number 0 is pushed onto the stack. Flow control operators interpret this value as False.
OP_TRUE/OP_1	0x51	The number 1 is pushed onto the stack. Flow control operators interpret this value as True.
OP_2 - OP-16	0x52-0x60	The number 2-16 is pushed onto the stack.

The following opcodes are used to implement flow control logic as well as for scripts to stop execution:

Opcode	Hex	Description
OP_IF	0x63	If the top stack value is OP_TRUE/OP_1 then the following statements are executed until OP_ENDIF or OP_ELSE is found.
OP_NOTIF	0x64	If the top stack value is OP_FALSE/OP_0 then the following statements are executed until OP_ENDIF or OP_ELSE is found.
OP_ELSE	0x67	If the preceding OP_IF, OP_NOTIF or OP_ELSE was not executed then following statements are executed. Multiple OP_ELSE sections can be executed as a result.
OP_ENDIF	0x68	Ends an if/else block. All blocks must end, or script execution fails. An OP_ENDIF without OP_IF earlier is also invalid.
OP_VERIFY	0x69	Marks transaction as invalid if top stack value is not OP_TRUE/OP_1. The top stack value is removed.
OP_RETURN	0x6a	Marks transaction as invalid.

The following opcodes are used to implement stack manipulation logic:

Opcode	Hex	Description
OP_TOALTSTACK	0x6b	Puts the input onto the top of the alt stack. Removes it from the main stack.
OP_FROMALTSTACK	0x6c	Puts the input onto the top of the main stack. Removes it from the alt stack.
OP_2DROP	0x6d	Removes the top two stack items.
OP2_DUP	0x6e	Duplicates the top two stack items.

OP_3DUP	0x6f	Duplicates the top three stack items.
OP_2OVER	0x70	Copies the pair of items two spaces back in the stack to the front.
OP_2ROT	0x71	The fifth and sixth items back are moved to the top of the stack.
OP_2SWAP	0x72	Swaps the top two pairs of items.
OP_IFDUP	0x73	If the top stack value is not OP_FALSE/OP_0, duplicate it.
OP_DEPTH	0x74	Puts the number of stack items onto the stack.
OP_DROP	0x75	Removes the top stack item.
OP_DUP	0x76	Duplicates the top stack item.
OP_NIP	0x77	Removes the second-to-top stack item.
OP_OVER	0x78	Copies the second-to-top stack item to the top.
OP_PICK	0x79	If the top of the stack is N. The item N position back in the stack is copied to the top.
OP_ROLL	0x7a	If the top of the stack is N. The item N back in the stack is moved to the top.
OP_ROT	0x7b	The 3rd item down the stack is moved to the top.
OP_SWAP	0x7c	The top two items on the stack are swapped.
OP_TUCK	0x7d	The item at the top of the stack is copied and inserted before the second-to-top item.

OP_SIZE	0x82	Pushes the string length of the top element of the stack (without popping it).

The following opcodes are used to implement bitwise and arithmetic logic:

Opcode	Hex	Description
OP_EQUAL	0x87	Returns OP_TRUE/OP_1 if the inputs are exactly equal, OP_FALSE/OP_0 otherwise.
OP_EQUALVERIFY	0x88	Same as OP_EQUAL, but runs OP_VERIFY afterward.
OP_1ADD	0x8b	1 is added to the input.
OP_1SUB	0x8c	1 is subtracted from the input.
OP_NEGATE	0x8f	The sign of the input is flipped.
OP_ABS	0x90	The input is made positive.
OP_NOT	0x91	If the input is 0 or 1, it is flipped. Otherwise, the output will be 0.
OP_0NOTEQUAL	0x92	Returns 0 if the input is 0. 1 otherwise.
OP_ADD	0x93	A and B are in the stack. A is added to B.
OP_SUB	0x94	A and B are in the stack. B is subtracted from A.
OP_BOOLAND	0x9a	If both a and b are not 0, the output is 1. Otherwise, 0.

OP_BOOLOR	0x9b	If a or b is not 0, the output is 1. Otherwise, 0.
OP_NUMEQUAL	0x9c	Returns 1 if the numbers are equal, 0 otherwise.
OP_NUMEQUALVERIFY	0x9d	Same as OP_NUMEQUAL, but runs OP_VERIFY afterward.
OP_NUMNOTEQUAL	0x9e	Returns 1 if the numbers are not equal, 0 otherwise.
OP_LESSTHAN	0x9f	Returns 1 if a is less than b, 0 otherwise.
OP_GREATERTHAN	0xa0	Returns 1 if a is greater than b, 0 otherwise.
OP_LESSTHANOREQUAL	0xa1	Returns 1 if a is less than or equal to b, 0 otherwise.
OP_GREATERTHANOREQUAL	0xa2	Returns 1 if a is greater than or equal to b, 0 otherwise.
OP_MIN	0xa3	Returns the smaller of a and b.
OP_MAX	0xa4	Returns the larger of a and b.
OP_WITHIN	0xa5	Returns 1 if x is within the specified range (left-inclusive), 0 otherwise.

The following opcodes are used to implement cryptographic operations:

Opcode	Hex	Description
OP_RIPEMD160	0xa6	The input is hashed using SHA-1.

OP_SHA1	0xa7	The input is hashed using SHA-1.
OP_SHA256	0xa8	The input is hashed using SHA-256.
OP_HASH160	0xa9	The input is hashed twice: first with SHA-256 and then with RIPEMD-160.
OP_HASH256	0xaa	The input is hashed two times with SHA-256.
OP_CODESEPARATOR	0xab	All the signature checking words will only match signatures to the data after the most recently executed OP_CODESEPARATOR.
OP_CHECKSIG	0xac	The entire transaction's outputs, inputs, and script (from the most recently executed OP_CODESEPARATOR to the end) are hashed. The signature used by OP_CHECKSIG must be a valid signature for this hash and public key. If it is, 1 is returned, 0 otherwise.
OP_CHECKSIGVERIFY	0xad	Same as OP_CHECKSIG, but OP_VERIFY is executed afterward.
OP_CHECKMULTISIG	0xae	Compares the first signature against each public key until it finds an ECDSA match. Starting with the subsequent public key, it compares the second signature against each remaining public key until it finds an ECDSA match. The process is repeated until all signatures have been checked or not enough public keys remain to produce a successful result. All signatures need to match a public key. Because public keys are not checked again if they fail any signature comparison,

		signatures must be placed in the scriptSig using the same order as their corresponding public keys were placed in the scriptPubKey or redeemScript. If all signatures are valid, 1 is returned, 0 otherwise. Due to a bug, one extra unused value is removed from the stack.
OP_CHECKMULTISIGVERIFY	0xaf	Same as OP_CHECKMULTISIG, but OP_VERIFY is executed afterward.

The following opcodes are used to implement time management operations:

Opcode	Hex	Description
OP_CHECKLOCKTIMEVERIFY	0xa6	Marks transaction as invalid if the top stack item is greater than the transaction's LockTime field; otherwise, script evaluation continues as though an OP_NOP was executed. The transaction is also invalid if 1. the stack is empty, or 2. the top stack item is negative, or 3. the top stack item is greater than or equal to 500000000 while the transaction's LockTime field is less than 500000000, or vice versa; or 4.
OP_CHECKSEQUENCEVERIFY		Marks transaction as invalid if the relative lock time of the input is not equal to or longer than the value of the top stack item

To demonstrate how scripts are executed, this is a script example:

```
OP_DUP OP_HASH160 <hash_public_key> OP_EQUALVERIFY OP_CHECKSIG
```

When encoded in hexadecimal, the transaction script will look as in Figure 18 when embedded into the block.

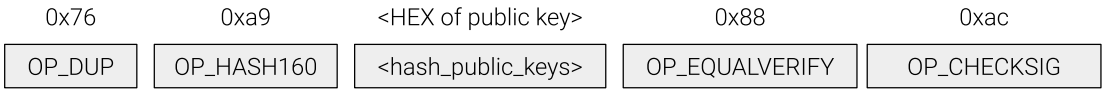

Figure 18 Transaction Script Encoding

The executing the script, the interpreter executes each of the instructions step-by-step. Figure 19 shows the contents of the stack at each step of the execution. This transaction execution will be considered valid as it ends with OP_TRUE on the stack.

Step	Stack state	Script	Description
1	<script_signature> <public_key>	OP_DUP OP_HASH160 <hash_public_key> OP_EQUALVERIFY OP_CHECKSIG	Initial state: scriptSig and scriptPubKey are combined.
2	<script_signature> <public_key> <public_key>	OP_DUP OP_HASH160 <hash_public_key> OP_EQUALVERIFY OP_CHECKSIG	Duplicates the top stack item.
3	<script_signature> <public_key> <hash_public_key>	OP_HASH160 <hash_public_key> OP_EQUALVERIFY OP_CHECKSIG	The input is hashed twice: first with SHA-256 and then with RIPEMD-160.
4	<script_signature> <public_key> <hash_public_key> <hash_public_key>	<hash_public_key> OP_EQUALVERIFY OP_CHECKSIG	<hash_public_key> is pushed onto the stack

5	<script_signature> <public_key>	OP_EQUALVERIFY OP_CHECKSIG	Continues if the two top elements of the stack are equal. Removes it from the stack.
6	<1>	OP_CHECKSIG	The entire transaction's outputs, inputs, and script are hashed. The signature used by OP_CHECKSIG must be a valid signature for this hash and public key. Pushes OP_TRUE onto the stack.

Figure 19 Step by Step script execution

6.3. Blockchain Initialization

Every network starts with at least one node. The first block of a network is referred to as the genesis block. It is the only block on the chain where the previous block hash is zero. Typically, the genesis block contains a single transaction that initializes the cluster. The creator of the first block signs establishes the cryptographic criteria to be met for subsequent transactions.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "0",
           "txid": "0xFFFF",
          "scriptSig": "Any text can go here"
      }
  ],
  "vout": [
      {
          "value": {
              "kind": "Cluster",
              "apiVersion": "v1",
              "metadata": {
                  "name": "Demo"
              }
```

```
          },
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_creator>
OP_EQUALVERIFY
OP_CHECKSIG"
      }
  ]
}
```

Figure 20 First cluster transaction

In the example provided in Figure 20, the operator explicitly requires the following transaction, i.e., using this one as input, to provide the operator's public key to be hashed and compared with the one embedded in the script (OP_EQUALVERIFY). This of course does not provide any security. For that, the script also requires the user to sign the transaction with the private (OP_CHECKSIG). Initially, the creator is the only one with access to the cluster's resources.

The next step toward making this cluster functional is to add more worker nodes to the network. Any node can participate in the network, but only nodes part of the cluster can host applications. For example, a node might connect to the network via the gossip protocol but until a new transaction is added to the blockchain adding the node, the node would not be considered a candidate to host applications by the scheduler. Non-cluster node examples include developer nodes that host applications on the cluster that can run a client application to validate and analyze the chain's current state. Other nodes can participate as validators to obtain a copy of the block to perform other functions such as security scanning, policy compliance, SLA, capacity planning, etc.

Adding a node is considered a change to the cluster configuration. Any transaction that modifies the cluster requires the latest Cluster transaction as input. The example in Figure 21 is a transaction adding a node to the cluster. The input of the transaction references the output of the transaction that created the cluster or any subsequent transactions that updated the cluster resource. The transaction's output is the node definition following the Kubernetes resource format, which includes the DNS name of the node.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the lastest cluster transaction>",
```

```
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "Node",
              "metadata": {
                  "name": "my-first-node",
                  "labels": {
                      "name": "development"
                  }
              }
          },
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

Figure 21 Transaction adding a node to the cluster

Nodes can optionally have a locking script that enables delegation of the operations of the node configuration. The transaction's output utilizes the same protection mechanism used by the cluster transaction. Any future transactions referring to the output of the node creation transaction must provide proof that they hold the private key corresponding to the public key embedded within the script by signing the transaction with it.

Once a transaction has been executed, it cannot be replayed. Nodes must check if the transaction has already been added to a previous block. Additional optimizations will be considered in the future to avoid the overhead of maintaining an index of all the transactions in the blockchain.

The example in Figure 21 requires that any future modifications to the node need either using the latest cluster transaction (parent container) or the latest transaction of the node. Figure 22 provides an example of a transaction to take the anode offline by updating the labels attached to the node. Once the transaction is added to the blockchain, future transactions must refer to the new transaction.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest node transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "Node",
              "metadata": {
                  "name": "my-first-node",
                  "labels": {
                      "name": "offline"
                  }
              }
          },
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

Figure 22 Transaction modifying an existing node

The transaction signature needs to be calculated using the public key of the admin of the node instead of the creator of the cluster. A node admin can also change the locking script to change the signature requirements for the node, including removing his permissions by removing the locking script with a subsequent transaction.

6.4. Adding a namespace

Namespaces provide a mechanism for isolating groups of resources like applications within a cluster. Namespaces require the latest cluster transaction as the input, including the signature that satisfies the locking script.

Namespaces are used to separate resources across different environments—for example, development, staging, and production. Figure 23 includes an example of a transaction that creates a new “development” namespace using the Kubernetes resource format definition.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest cluster transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "Namespace",
              "metadata": {
                  "name": "default",
                  "labels": {
                      "name": "development"
                  }
              }
          },
          "scriptPubKey": "
OP_DUP
```

```
OP_HASH160
<hash_public_key_from_creator>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

Figure 23 Transaction creating a new namespace

As in the previous example with nodes, namespaces can optionally have a locking script that enables other operators to access the resources that would require the Namespace transaction as input. It is also possible to use scripting capabilities to add complex access logic through the locking script. Figure 24 is an example of transactions that can be done by multiple administrators.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest cluster transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "Namespace",
              "metadata": {
                  "name": "default",
                  "labels": {
                      "name": "development"
                  }
              }
          },
          "scriptPubKey": "
```

```
1
<hash_public_key_owner_1>
<hash_public_key_owner_2>
<hash_public_key_owner_3>
3
CHECKMULTISIG"
      }
  ]
}
```

Figure 24 Namespace with multiple administrators

6.5. Complex Locking Logic

It is also possible to restrict within a transaction the conditions under which future transactions can access a resource. The following are examples of common administrative restrictions:

6.5.1. Granting permissions to one or many users for five days

```
5d
OP_CHECKSEQUENCEVERIFY
OP_VERIFY
1
<hash_public_key_owner_1>
<hash_public_key_owner_2>
<hash_public_key_owner_3>
3
OP_CHECKMULTISIG
```

The 5d constant estimates the number of blocks that can be generated in five days. OP_CHECKSEQUENCEVERIFY checks whether the current block number is more than the block number of the input transaction plus the number on the stack, in this case, five days' worth of blocks.

6.5.2. Locking a resource for some time

```
5d
OP_CHECKSEQUENCEVERIFY
```

```
OP_DROP
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG
```

6.5.3. Require multiple signatures. For example, 2 out of 3 operator users need to sign the transaction

```
2
<hash_public_key_owner_1>
<hash_public_key_owner_2>
<hash_public_key_owner_3>
3
OP_CHECKMULTISIG
```

6.5.4. Require two signatures for some time. Then after that period, any modifications can be done with a single admin signing the transaction

```
OP_IF
  2
  <hash_public_key_owner_1>
  <hash_public_key_owner_2>
  <hash_public_key_owner_3>
  3
  OP_CHECKMULTISIG
OP_ELSE
  5d
  OP_CHECKSEQUENCEVERIFY
  OP_DROP
  1
  <hash_public_key_owner_1>
  <hash_public_key_owner_2>
  <hash_public_key_owner_3>
  3
  OP_CHECKMULTISIG
OP_ENDIF
```

The previous examples of locking logic apply to any type of resource, including nodes and clusters. This enables scenarios where specific resources have lesser operational requirements while others, like clusters, enforce more complex logical conditions. It would be common to require multiple signatures to perform any changes at the cluster level.

6.6. Deleting Resources

Resources can be deleted by specifying the latest output transaction that modified the resource as input. The output of a transaction that deletes a resource is null. It is also important to note that all descendants are deleted when deleting a parent resource. This means that nodes need to keep an index that maps resources to the last transaction that modified the resource. Figure 25 is an example of decoded transaction deleting a namespace.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest namespace
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
  ]
}
```

Figure 25 Delete resource transaction

Clusters are the only resources that cannot be deleted, as this is the root of all transactions.

6.7. Multiple transaction outputs

A transaction can have multiple outputs. For example, creating or updating multiple namespaces within a single transaction is possible. Every output of the transaction has a locking script. Figure 26 is an example of a transaction with more than one output namespace resources. Because each resource needs to be uniquely identified as an input of future transactions, creating multiple resources in the same output is not permitted. Future research will evaluate utilizing different indexing mechanisms for transaction outputs to provide a more compact representation for more than one resource.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest cluster transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "Namespace",
              "metadata": {
                  "name": "development-ns",
                  "labels": {
                      "name": "development"
                  }
              }
          },
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_creator>
OP_EQUALVERIFY
OP_CHECKSIG",
      },
      {
          "value": {
              "apiVersion": "v1",
              "kind": "Namespace",
              "metadata": {
                  "name": "production-ns",
                  "labels": {
                      "name": "production"
```

```
                    }
                }
            },
            "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_creator>
OP_EQUALVERIFY
OP_CHECKSIG",
        }
    ]
}
```

Figure 26 Multiple output transaction

When a transaction input refers to a transaction with multiple outputs, it is necessary to use the "vout" field for the correct output. Figure 27 shows a transaction deleting the production namespace created on the previous transaction.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the multiple ns transaction>",
          "vout": 1,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ]
}
```

Figure 27 Deleting resources from multi-output transactions

6.8. Permission hierarchy

Access to resources is determined by the locking script used in its latest transaction that modified the resource or the locking script of any parent resources in that order. When evaluating whether a transaction should be allowed, the depth-last order in the hierarchy in Figure 28 is applied.

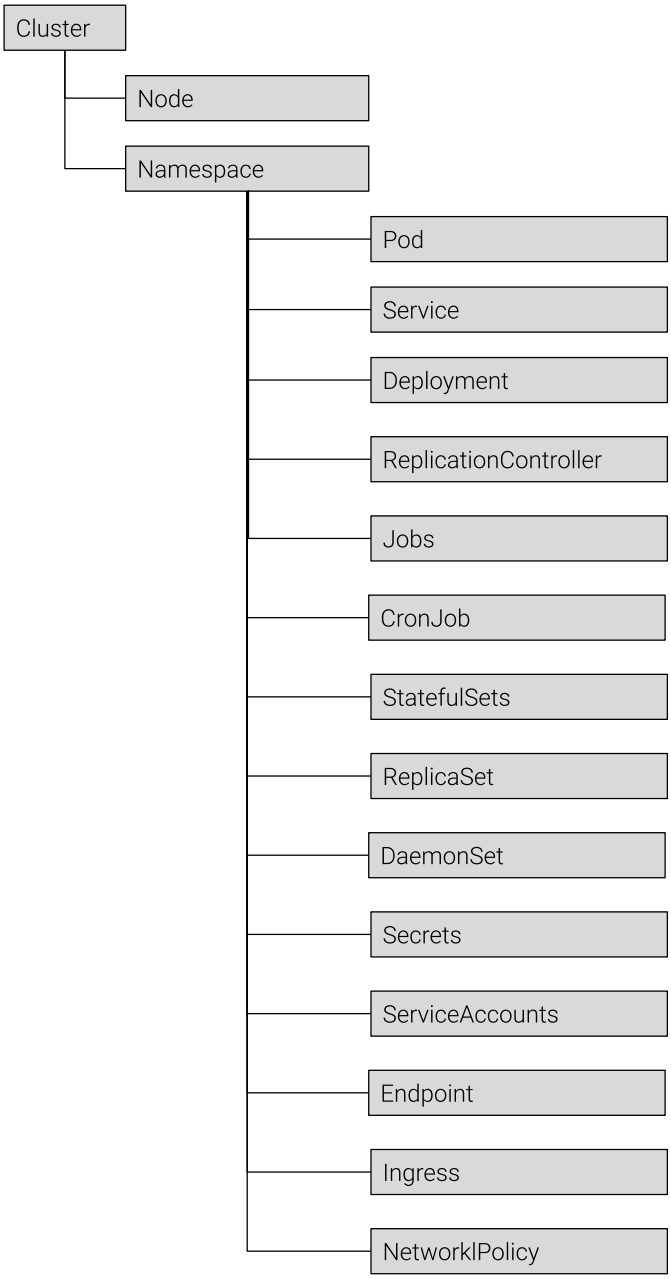

Figure 28 Resource hierarchy

For example, a Pod transaction can include a locking script that references the hash of Alice's key. Bob's key hash is included in the script transactions that locked the namespace where the Pod was created. Even if the new Pod transaction does not include Alice's key hash, he can still modify the resource by using the transaction of the namespace as input to alter the Pod.

The process to evaluate of access control follows these steps:

1. Check that the latest input transaction modified the output resource specified in the “vout” field.
2. If the transaction's output is the object being modified, evaluate the locking script. If the locking script returns OP_TRUE, accept the transaction.
3. If the transaction's output is another object, check whether the object is the parent of the object being modified. If it is the case, evaluate the locking script. If the locking script returns OP_TRUE, accept the transaction.

CHAPTER 7. BUILDING THE NETWORK

7.1. Nodes joining the network

Nodes connect to a network via the P2P gossip protocol. When a node is first initialized, it needs to be configured with one or more nodes from which it can discover the other nodes on the network.

When a node starts up for the first time, it connects to the configured node address resolved by the DNS service. If the node connects to the network for the first time, it will not know the chain or other peers. After a successful connection, the node calls the GetPeers function on the other nodes to receive the list of known peers to be persisted in the peer's database.

To participate in the network operations, a node must first download the network's blockchain. By calling GetBlocks on the peer nodes, it will receive the longest known chain by its peers in the form of block headers. Once the list of blocks is parsed, it gets added to the list of pending blocks and proceeds to request those blocks across all known peers.

By looking at the block headers, the node has enough information to validate the chain by looking at the hash of each block. It does not know, however, whether the transactions on the block are valid.

After the node downloads the largest known chain, it parses the contents and builds the state database with the information by processing each block in order and validating the transactions. It also validates the list of available peers and updates it to ensure it only accepts proposed blocks from peer nodes part of the cluster during that process.

While a node can only receive new proposed blocks in the form of block headers from other nodes in the cluster, it can download Blocks from any other node connected to the network. After a block is downloaded, it is validated through the Merkle tree hash stored in the block header. If a peer provides an invalid block, the peer is marked as blacklisted.

Downloading the chain and building the state database is expensive as it requires validating every single transaction in the blockchain. A possible optimization is to provide a parameter that establishes a recent block hash as confirmed valid as part of the node's configuration, therefore eliminating the need to validate every single transaction in the block. Another possible optimization is including the hash of the configuration resource tree in every block, enabling a new node to download the tree from another node and validating the hash, further accelerating the initialization process.

To prevent Sybil attacks, the node should only maintain a percentage of node connections to non-cluster nodes and transmit the list of black-listed nodes when other peers call GetPeers on the node. When a node gets added to the cluster through a

transaction, the node should add the node's address to the peer list. Incoming connections from nodes outside the network should be limited to ensure that non-cluster nodes use all network connections.

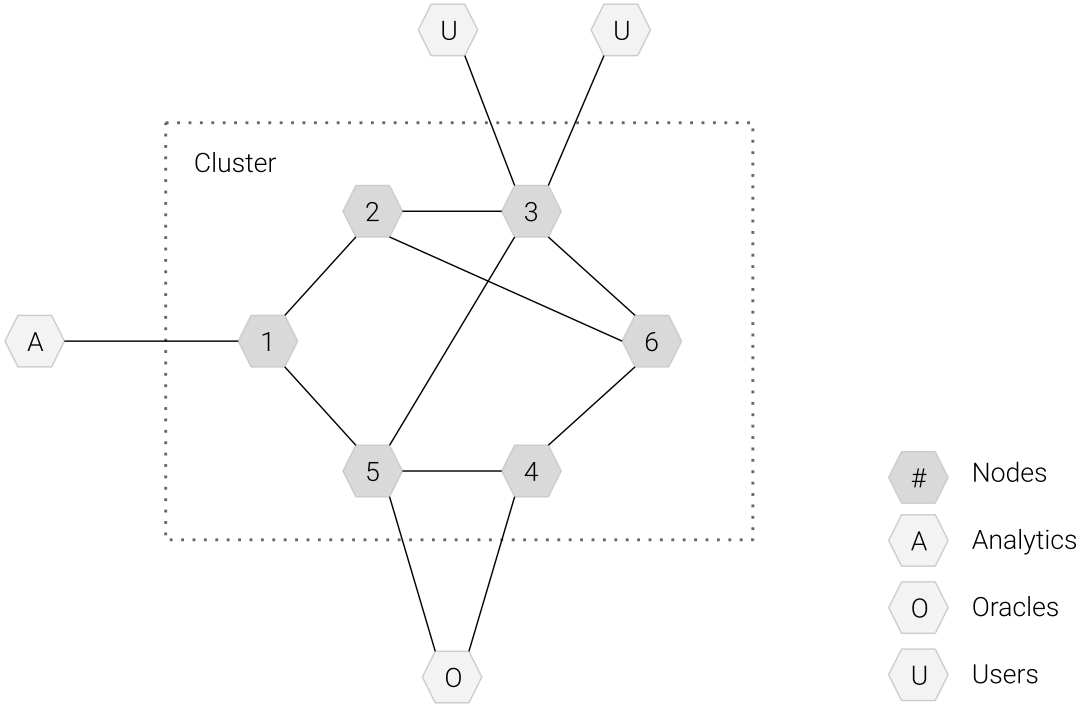

Figure 29 Node network topology

Non-cluster nodes are allowed to obtain a copy of the blockchain for different use cases:

1. Development: A developer needs a copy of the blockchain to build and test transactions before submitting them to the cluster. Users submitting transactions need access to the blockchain to calculate signatures and hashes of resources being used from previous transactions. In addition, testing and locking scripts are critical operations as transactions in the cluster are irreversible.
2. Analytics: Building systems that aid with the analytical operations of the cluster. For example, capacity planning, cluster utilization, efficiency, consensus performance, security analytics, etc.
3. Watchtower: Watchtowers act as ‘watchdogs’ of the blockchain to identify and penalize malicious nodes. For example, monitoring blocks' contents and removing nodes generating blocks that do not contain transactions

broadcasted through the network, indicating the node might have been compromised.

4. Oracles: Oracles are auxiliary input nodes that produce transactions that help with the operations and maintenance of the network, for example, by creating transactions that report the application's state. Oracle inputs are signed and support the construction of locking scripts. For instance, an Oracle can provide a transaction signature that unlocks a resource that needs to be updated to scale up resources. Oracles are helpful to gate-keep processes based on data, not just signatures.

7.2. Submitting a transaction

Users can submit transactions directly to nodes through the RPC API. When a node validates transactions as soon as they are received, including:

1. The input transaction exists.
2. The input resource is valid, including parsing the content and ensuring well-formed.
3. The input resource has a locking script.
4. The transaction used to refer to the input resource is the latest transaction in the chain for that resource.
5. The SigScript field satisfies the inputs of the locking script and, upon execution, returns OP_TRUE.
6. The output resources are valid.
7. The output script, if any, is valid.

If all checks pass, the transactions are added to the Pending changes transaction list and communicated to all other nodes connected to that node. Notice that the transaction can still become invalid in the context of a new block due to another transaction invalidating it.

7.3. Validating a transaction

When a user submits a transaction, two possible mechanisms satisfy the validation requirements—direct or indirect resource access.

- Direct access transactions reference the latest transaction with the target object as output and index the correct output. It's the user's responsibility to identify the most recent transaction and produce a signature that meets the requirements of the locking script.

- Indirect access transactions have as input the latest transaction of the parent resource. For example, creating a node refers to the output of the most recent cluster resource transaction.

As previously stated, locking scripts are simple, stack-based, and processed from left to right as a series of sequential instructions. Data is pushed onto the stack, and opcodes are used to perform operations on the items on the stack. Only resources that have output scripts can be used in subsequent transaction inputs.

7.4. Block Formation

Validators check every transaction during the block forming process. A node is formed by ordering the pending transactions and encoding them into a single block. The selection of transactions to be included in a block is a critical aspect of the system's design and will be further analyzed in the following sections. Transaction order is performed through both Topological and Canonical Ordering.

- Topological ordering by ordering transactions according to the position of the input resources in the permission hierarchy. In other words, placing transactions in the following order: first clusters, then nodes, namespaces, etc. Transactions with the same input are ordered using the same process but evaluating the outputs, placing null output first. For example, a transaction that creates a node is placed before a transaction that creates a namespace, as both would have the same input.

- Canonical ordering occurs when two resources have equivalent inputs and outputs in the resource permission hierarchy. When this happens, those transactions are ordered by the transaction Id calculated as the SHA256 of the transaction data. For example, two transactions that create a node will be shorted by the transaction ID.

Topological ordering ensures that sequential transactions that depend on a previous transaction are evaluated in a way that maximizes transaction validity. For example, a pod cannot be created until the parent namespace is created. Null output ordering is essential for cases when a transaction that eliminates resources needs to be processed in the same block as a transaction that allocates new resources.

Canonical ordering ensures that the output is unique and deterministic given the same set of transactions. In other words, given the same unordered transactions, the result after ordering would be the same regardless of who performs the ordering or when the operation is done.

CHAPTER 8. CONSENSUS ALGORITHMS

8.1. Selecting a block creator

Minting a block is the most critical operation in a blockchain. The following section will analyze different algorithms that can be used to ensure that blocks are minted, validated, and added to the blockchain throughout the network while minimizing the amount of trust required. In essence, these algorithms enable the capability to achieve consensus on which blocks to add to the chain based on rules that ensure fairness and security to all participants.

The analysis we will perform include, when relevant, their characteristic behavior for the following properties:

1. Partition resistance: The ability to operate and recover when network partitions occur for a short time or more extended periods. Including single network partitions, multiple network partitions, and network partitions with Byzantine Agents. In this section, we will also analyze the behavior of the network under different latency scenarios.
2. Resource Consumption: The total amount of resources used to mint a new block and distribute it across the network and the total amount of resources required to tamper with a block in the chain at different depth levels
3. Byzantine fault tolerance: What happens when an actor decides not to follow the rules and tamper with blocks of the chain or newly minted blocks. Including when the BFT notes are the minority and the majority. The analysis will include normal circumstances and network partition scenarios.
4. Availability: What are the necessary conditions for a transaction to be submitted to the network and added to a block with a reasonable guarantee of not being rolled back. In other words, the transaction is statistically confirmed.

8.2. Consensus algorithms

The following consensus algorithms will be evaluated as part of this research:

1. Proof of work (PoW) is a consensus algorithm based on demonstrable computational effort across a fixed time window, forcing each party to upfront a total energy/computational cost proportional to their weight on the consensus effort.
2. Proof of space (PoS) is a consensus algorithm based on demonstrable storage capacity requiring every participant to pre-compute and store an established function output. Participants must be able to prove knowledge of that output at any time, ensuring a commitment to integrity by upfronting the storage cost.
3. Proof of Authority (PoA) is a consensus mechanism based on the proven identity of the participants. This algorithm requires establishing a level of trust across the participants.
4. Proof-of-stake (PoS) is a consensus algorithm based on demonstrable funds at stake requiring all participants to deposit a monetary amount in an escrow account controlled by a cryptographic protocol.

As part of the evaluation, we will consider the following security attacks:

1. Distributed Denial of Service Attack: Overwhelming the system with transactions, for example, using compromised keys.
2. Sybil Attack: Overwhelming the system with Byzantine validator nodes or compromised nodes submitting incorrect validations or block submissions.

8.3. Proof of Work

Proof of work (PoW) is a cryptographic proof in which one party proves to others that a certain amount of a specific computational effort has been expended. This section analyzes how adding PoW requirements to the consensus algorithm ensures a homogeneous selection of nodes generating blocks and secures the network by guaranteeing that any attacker compounds the computational requirements required to disrupt the process.

As the Application Specific Integrated Circuits (ASICS) and Field Programmable Gate Arrays (FPGA) are used for PoW computation for blockchain consensus operation, the decentralized nature of blockchain networks is being threatened. For simplicity, we will focus on SHA256 proof of work. The need for more complex solutions to protect the network from adversaries with hardware-accelerated devices should be noted.

In PoW, every node competes in building a block by iterating through computing the block header's hash combined with a Nonce. Before a hash can be calculated, the block needs to be formed. In PoW consensus, there is a delay between when a transaction is submitted and when the transaction is added to the block. The race to build and hash the next block starts after a node has received a valid block, meaning all transactions in the block are correct, and the block's hash meets the target difficulty levels.

8.4.1. Building new blocks

Nodes gather pending transactions from the pending transaction pool to form new blocks. After ordering and validating them, transactions are hashed and organized into a Merkle tree. The tree's root is added to the block header, which will be hashed as part of the mining process. The node then can start iterating through hashing of the block header by adding the current value of the node Nonce and incrementing it in every iteration.

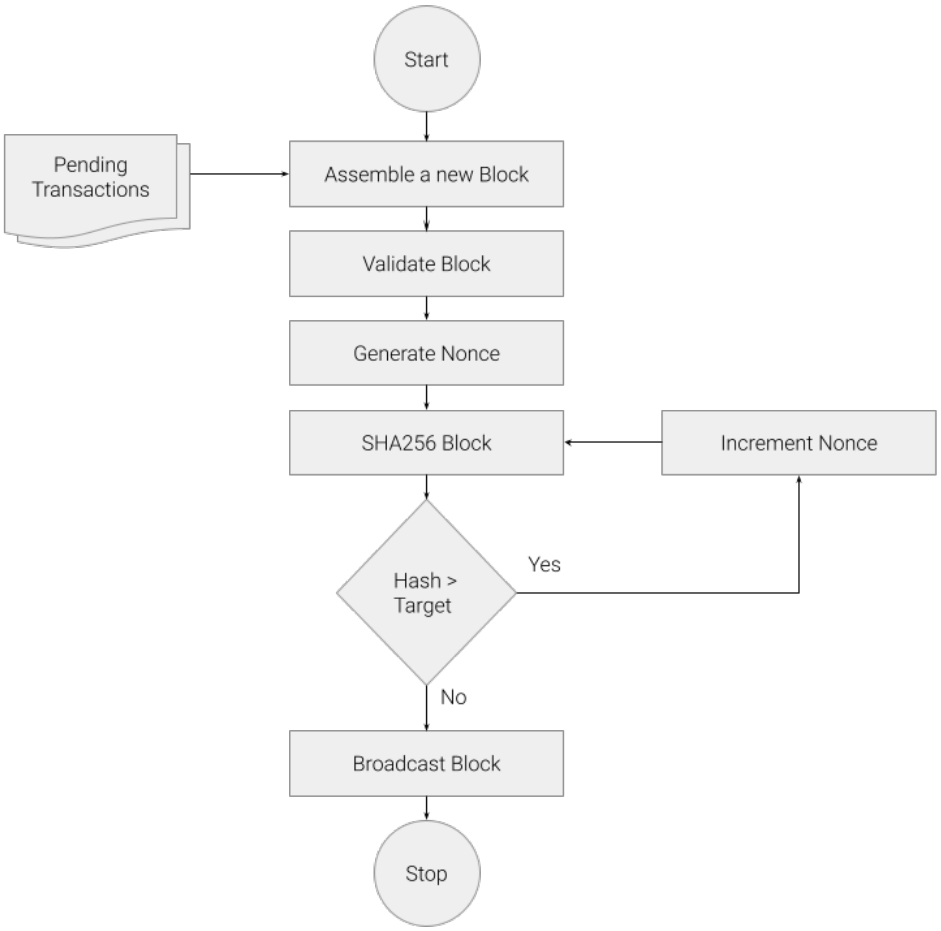

Figure 30 Proof of Work consensus

The Nonce of the node is an integer value initialized when the node is created and maintained throughout the node's lifetime. Because Nonce is persisted and not reset on every block iteration, it ensures random distribution of Nonces across the network over time. For example, if two nodes with identical computing power were created simultaneously and started with the same Nonce value, network timing differences and finding hashes that meet the target requirement will diverge the Nonces of those nodes over time.

8.4.2. Target difficulty

The Target value is determined by the current network difficulty setting, which carries over across blocks. Nodes are tasked to find a 256-bit unsigned integer whose hash combined with the header must be equal to or below for that header to be a valid part of the blockchain. Because hashes are randomly distributed, the average time required to find a hash meeting the difficulty requirements is proportional to the network's computing power.

In essence, the target is inversely proportional to the difficulty. The difficulty is encoded as a compact representation of a 256-bit number. The first byte of the 32-bit field represents an exponent, and the remaining 3 bytes encode a mantissa.

$$TargetDifficulty = Mantissa * 2^{8*(exponent-3)}$$

At the genesis of the blockchain, the first difficulty target can be inferred by estimating the hashing performance of the first node and applying the following formula.

$$Difficulty = \frac{MaxTargetHashPerSecond}{HashPerSecond * TargetBlockTime}$$

The MaxTarget is the highest value of a 248-byte number. In this case:

```
0x00000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

Because Bitcoin stores the target as a floating-point type, this is truncated:

```
x00000000FFFF0000000000000000000000000000000000000000000000000000
```

Which is the compact representation being 0x1D00FFFF.

TargetBlockTime is the desired number of seconds between blocks. This parameter will determine block latency and should be carefully selected. It should be high enough to minimize the chances of 2 or more nodes generating a block simultaneously yet low enough to meet operational latency requirements.

As more nodes are added to the network, the difficulty level must be adjusted to ensure that the network produces blocks at the desired TargetBlockTime. Difficulty adjustments are calculated automatically according to the height of the block:

$$NextDifficulty = \frac{CurrentDifficulty * TargetBlockTime * BlockAdjustmentPeriod}{\sum_{n=1}^{BlockAdjustmentPeriod}(Timestamp_n - Timestamp_{n-1})}$$

BlockAdjustmentPeriod is a network parameter that specifies the number of blocks between adjustments. Given that this is not a complex operation to calculate, the value can be estimated by taking into consideration the rate of change of total available hashing power, which should be targeted to be a percentage of the entire computing capability of the network to ensure that the threats to the consensus algorithm scale with the size of the network.

8.4.3. Block collisions

Two nodes may find a solution to the hash problem simultaneously or within a period where the broadcasting of the block happens simultaneously. Every node must follow the rules when receiving a valid block:

1. If the block is the next one in the chain, add it to the chain.
2. If the block number is higher than the next block in the chain, request any blocks that precede the received block and adopt the sub-chain.
3. If blocks in the current chain collide with the sub-chain, discard those blocks and adopt the new chain. Transactions that were part of the sub-chain and are not found in the new chain are re-added to the pending transaction list.

While it might be possible that the blockchain is forked temporarily by nodes adopting different versions of blocks, these rules ensure that the chain self-stabilizes by ensuring the survival of the longest chain generated by the chain with the largest group of nodes that adopted it. If the network is split exactly in half, network timings and performance invariance will ensure that one of the chains falls behind and deprecates. As illustrated in Figure 31 the same logic applies if the collision occurs between more than two nodes.

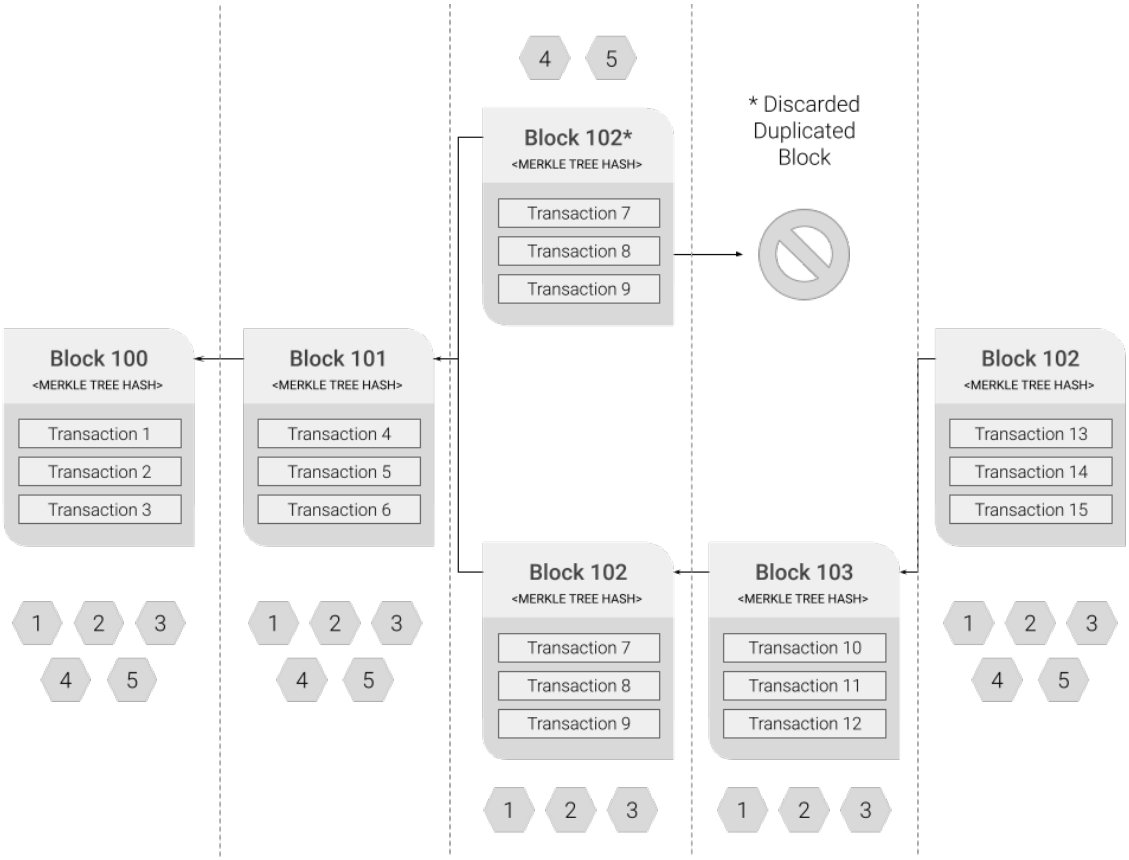

Figure 31 Resolving chain splitting.

The total resource consumption of the consensus algorithm is the sum of all computing power not used by the network to execute applications. While searching for the next block hash is not considered productive work, it has several security benefits and incentives for the network owners to right-size the network to maintain an acceptable level of unproductive resource consumption.

8.4.4. Byzantine Fault Tolerance

PoW networks are Byzantine fault-tolerant to a certain extent. In case some network nodes are compromised, the consensus algorithm can still be effective if the attacker controls less than 51% of the hashing power. Those nodes can be instructed to ignore legitimate transactions and produce blocks with valid but irrelevant transactions. In addition, compromised nodes must not accept legitimate valid blocks for the attack to be effective. The security attack will stop the processing of legitimate transactions and prevent operations on the network, which can be potentially used to disrupt applications running in the cluster.

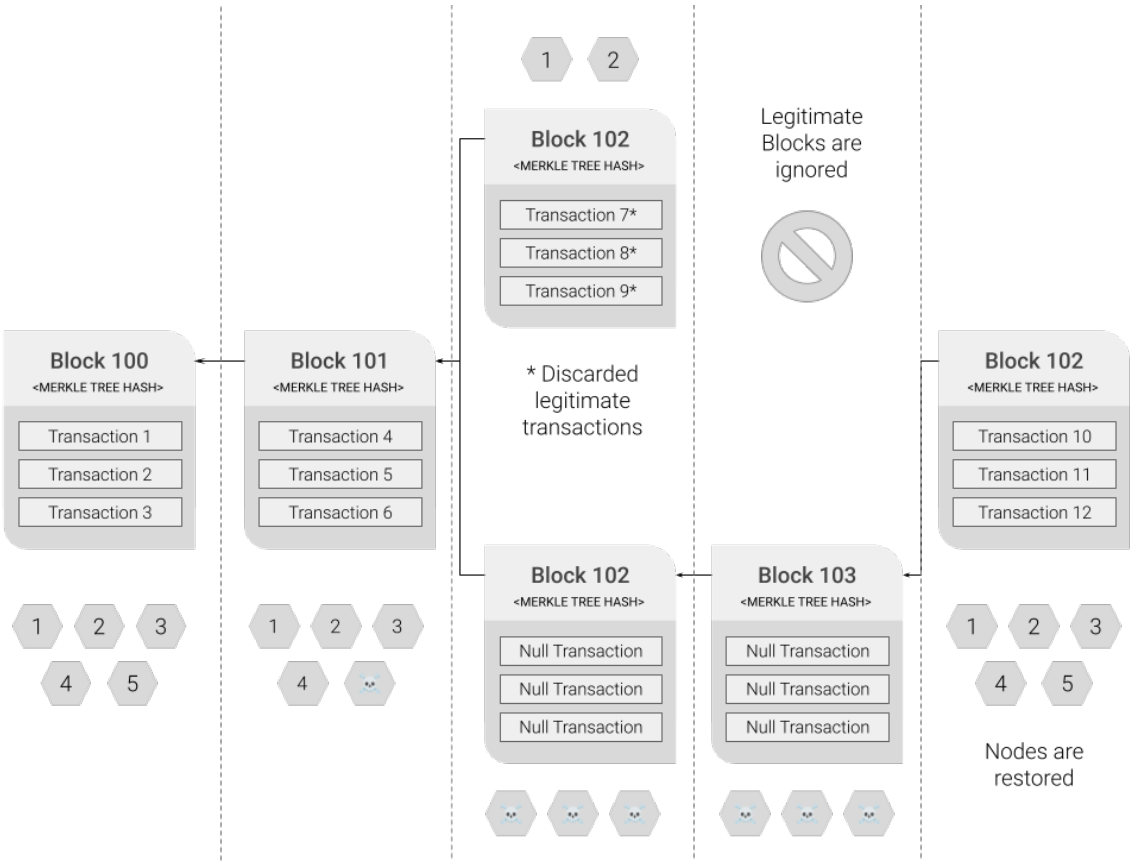

Figure 32 Blockchain 51% attack

Suppose the attacker compromises less than 51% of the hashing power. In that case, it can still disrupt operations but not halt them. The network will still accept valid blocks generated by normal nodes, and compromised nodes will be forced to take the blocks to ensure they can follow the longest chain to broadcast empty blocks.

Table 4 Hashing power attack ratios contain an example of the number of nodes that need to compromise the consensus process assuming 80% utilization of the network computing power for production workloads and 100 KH/s per node. Assuming Byzantine nodes utilize 100% of the compute power for hashing.

Legitimate Nodes	Compromised Nodes	Hashing Power	Compromised Hashing Power	% Of Dummy Blocks	% Of Legitimate Blocks
100	0	2 MH/s	0 MH/s	0%	100%
99	1	1.98 MH/s	100 KH/s	5%	95%

95	5	1.9 MH/s	500 KH/s	21%	79%
90	10	1.8 MH/s	1 MH/s	36%	64%
85	15	1.5 MH/s	1.5 MH/s	50%	50%
80	20	1.6 MH/s	2 MH/s	55%	Discarded
50	50	1.6 MH/s	5 MH/s	75%	Discarded

Table 4 Hashing power attack ratios

In the Table 3 example, when the attacker compromises less than 15 nodes, it can slow down the operational network throughput by producing valid dummy blocks added to the chain as if they contain legitimate transactions. In this case, the attacker does not have enough hashing power to create blocks to maintain a chain of dummy blocks fast enough. Once the hashing power crosses the 51% threshold, the attacker can, on average, generate more blocks than the rest of the network, therefore, ensuring the dummy chain is the one that survives, and any other block is discarded.

To restore partial control of the cluster, it is only necessary to restore enough nodes such that there are more legitimate nodes than compromised nodes. After system control is restored, all blocks generated by the byzantine nodes will still be part of the chain as those changes are irreversible.

It is possible to implement heuristics for legitimate nodes to reject blocks generated by potentially compromised nodes. Those techniques are outside the scope of this research.

8.4.5. Network Partitioning

Network partitioning occurs when a group of isolated nodes cannot communicate with the rest of the network's nodes. This is a common scenario when those nodes are not in the same data center, or the data center is partitioned into two or more availability zones.

When a network partition occurs, there is a risk that transactions submitted to the network partition with the shortest chain are lost once the network connectivity is restored.

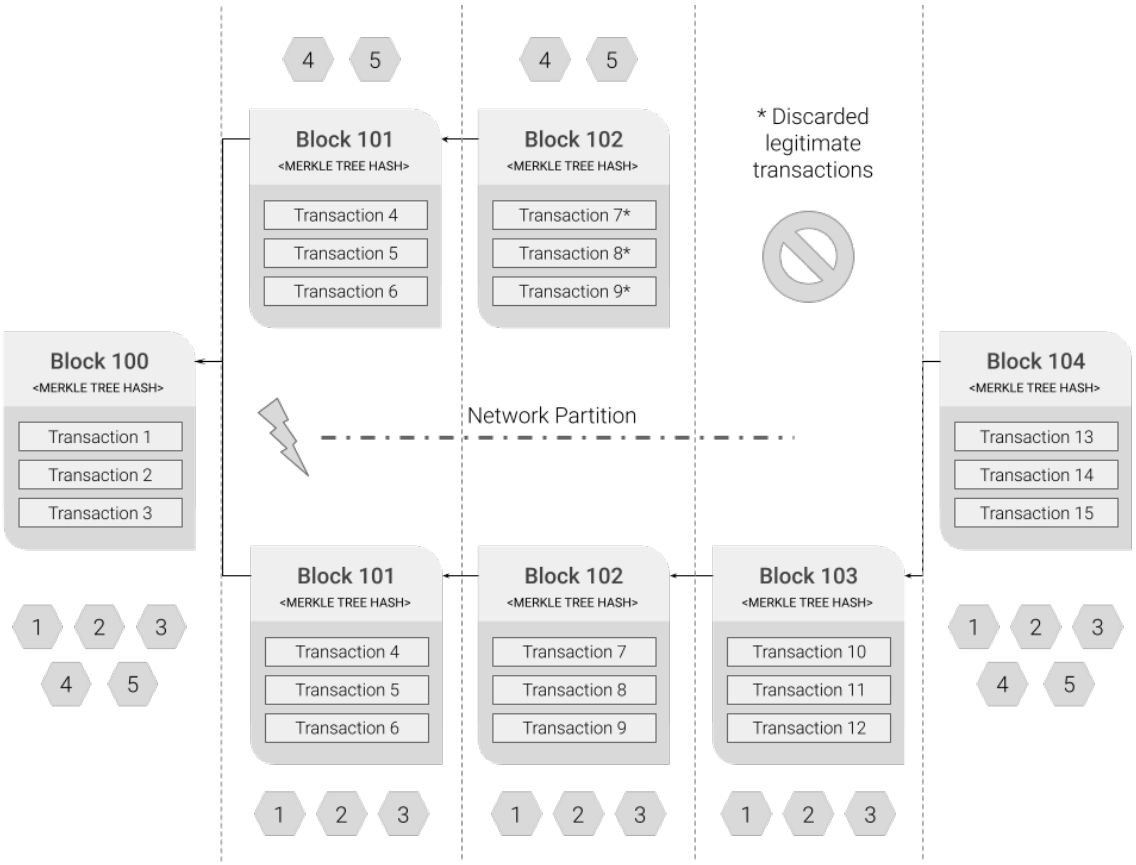

Figure 33 Network Partition in PoW

If Byzantine agents attack a network partition, each network partition will behave as if it was a unique cluster of nodes. In other words, if the byzantine agents in one network partition obtain more hashing power than the legitimate nodes, no new transactions will occur on that partition. Once the network partition is restored, the longest chain produced (the network partition with more hashing power) will spread across the cluster, and any other chains will be discarded.

In the example below, the network is partitioned into two. Assuming the same constraints as before of 80% productive utilization of legitimate nodes, the partition with 18 nodes and a total of 1.8 MH/s will be able to maintain a longer chain. However, the network partition with only two nodes and one of them compromised will not be able to process any new transactions as the byzantine node will have the ability to hash at 100 KH/s vs. 20 KH/s of the legitimate node. In addition, any transaction between the time the partition occurs and the node is compromised will be discarded once the longest chain is adopted.

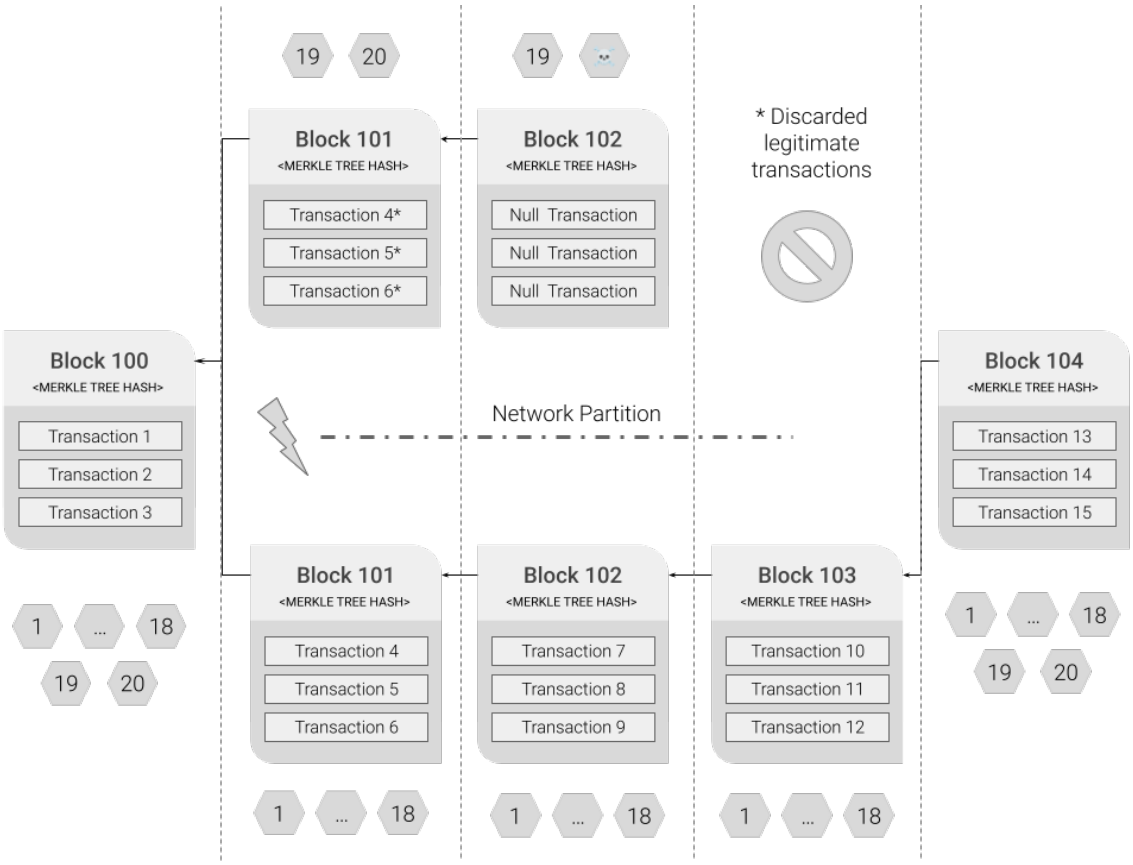

Figure 34 Restoring Network Partitions

8.4.6 Block size considerations

Block size, by default, can be up to 2^{32} bytes or 4Gb. Assuming an average transaction of 1024 bytes, the system will be limited to 4 million operations per block. It is important to note that changes that require more than 4GB of block space will be needed to be spread across multiple blocks. For example, a global configuration change to all resources in the cluster might require extending changes through numerous blocks. In addition, propagation of those transactions might not happen within the BlockPeriod time, therefore not reaching the node winning the hash lottery.

The following formula defines the throughput of the system:

$$TransactionThroughput = \frac{2^{32}}{TransactionSize * BlockPeriod}$$

If the BlockPeriod is 60 seconds and the average transaction size is 1024 bytes, the total average throughput of the cluster is 69905 operations per second.

8.4.7. Other Attacks

A Sybil attack is where an attacker tries to create as many nodes as possible to overwhelm the P2P network. Because nodes need to be explicitly added to the network, in other words, it is not a permission-less network, this type of attack is not possible. Nodes should connect to nodes that are not part of the cluster.

Distributed Denial of Service Attack (DDoS) occurs when a key that provides access to resources in the cluster is compromised, therefore enabling the possibility of submitting valid transactions to overwhelm the P2P network. Several countermeasures can be implemented to mitigate the effects of these attacks:

1. Transaction throttling: Establishing the maximum number of transactions that a node can emit or receive from another node or user.
2. Resource type quotas: Establish a maximum number of operations per input resource type.
3. Anomaly detection: Develop pattern matching algorithms that detect anomalous transaction operations.
4. Multi-signature locks: Requiring multiple signatures for operations reduces the risks when a signature is compromised.
5. Watchtowers: Require every transaction signed by a third party to ensure that the watchtower requirement is implemented for subsequent transactions and other external verifications to guarantee that only legitimate transactions are added to the network.
6. Time-locked transactions: Only accept transactions that impose a time lock after several transactions per second on the same resource have been executed.

One of the advantages of locking scripts is integrating with external systems via the encoding of complex logic and signature proofs.

Other types of DDoS attacks are possible by, for example, overwhelming the TCP/IP layer and blocking the ability of a node to connect to other nodes. These attacks are out of the scope of this research.

In large-scale distributed systems, Nonce distribution should be homogeneous due to the implicit randomness of the block generation process. In cases where a group of nodes colludes to take control over the network, it is beneficial for the attacker to sequence the Nonce hash calculation to minimize overlaps between attacking nodes. This manipulation of the Nonce could be detected by analyzing the Nonce distribution across the nodes.

An example of distribution manipulation happens when hash computing power is aggregated in mining pools. Figure 30 showcases the Nonce distribution of popular cryptocurrencies that utilize PoW and allows the aggregation of nodes via mining pools which sequence the Nonce being evaluated across all nodes part of the pool.

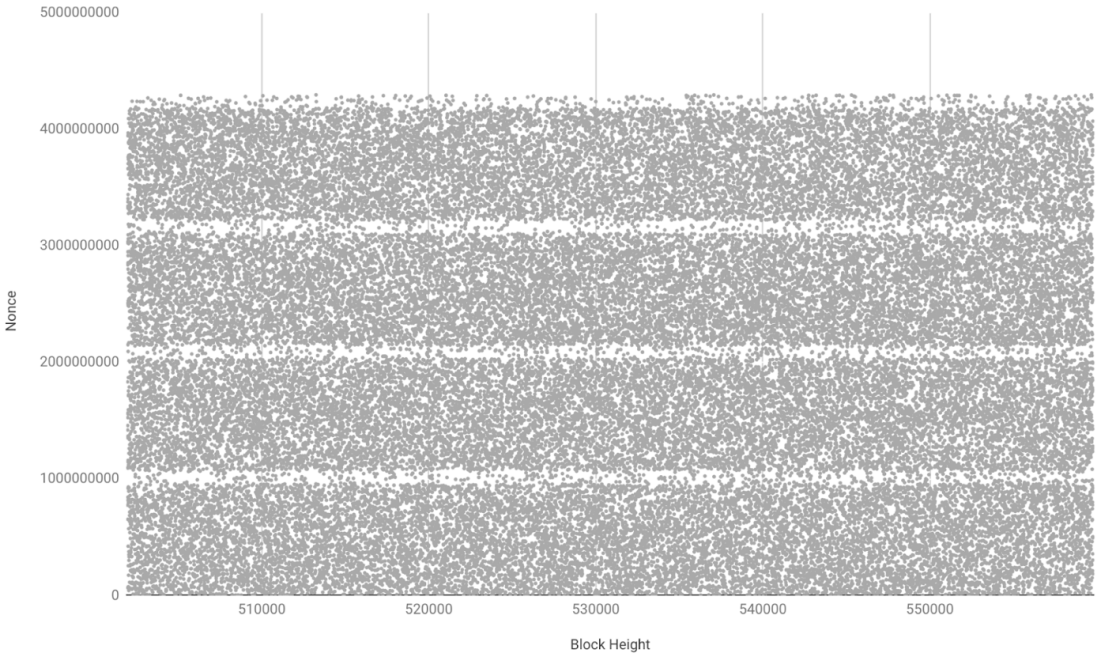

Figure 35 Nonce distribution across blocks

In cases where compromised nodes hashing power utilization is maximized, it would be possible to detect the increased hashing power of the network by detecting the accelerated production of both legitimate and dummy blocks.

8.5. Proof of Space

Proofs of Space (PoSpace) is very similar to proofs of work (PoW), except that instead of computation, storage is used to prove that the prover has reserved a certain amount of space. PoSpace is different from memory-hard functions in that the bottleneck is not in the number of memory access events but the amount of space required. If a prover does not reserve the claimed amount of space, it should be hard to pass the verification for security challenges. An adversary who stores a file of size significantly less than N bits should not be able to produce valid proof for a randomly selected challenge [65].

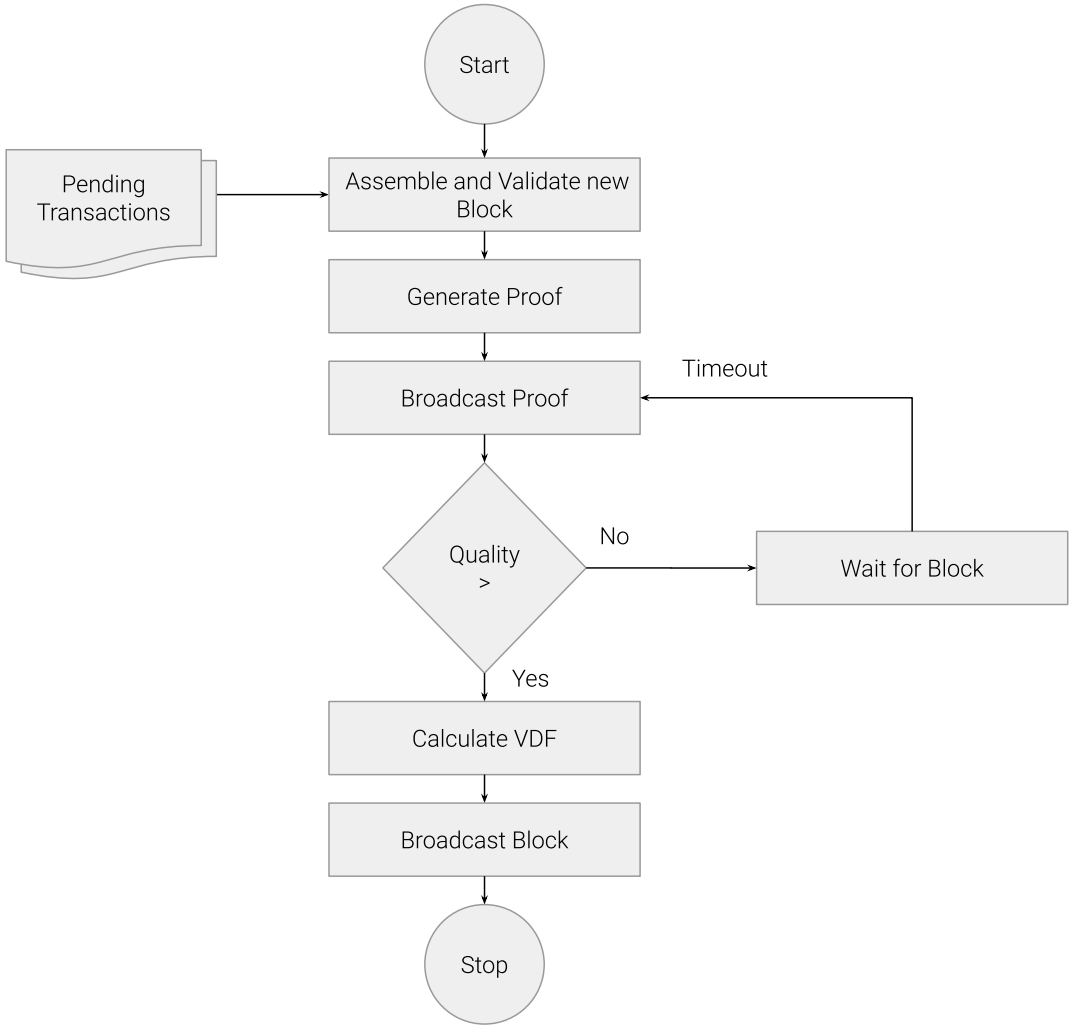

Figure 36 Proof of Space consensus

8.5.1. Building new blocks

Using a proof of space system, every node can immediately compute a proof, so we somehow need to nominate a node that will produce the next block. A possible implementation is to order the hash of the proof. The node producing the lowest scalar value of the proof hash should be the next block of the chain. To finalize the block, one must augment the block with the output of a verifiable delay function that limits the network's speed to generate blocks.

When a node is initialized, the PoSpace storage needs to be initialized. One of the advantages of this consensus protocol is that this is a single-time operation, reducing the total cost of resources to produce proofs. In addition, once the node is decommissioned, the storage space can be reclaimed. Figure 36 shows the process each node follows during the consensus mechanism.

Legitimate	Compromised	% Of Dummy	% Of Legitimate

Nodes	Nodes	Blocks	Blocks
100	0	0%	100%
99	1	5%	1%
90	10	90%	10%
50	50	50%	50%
49	41	100%	Discarded

Table 5 Proof of Space effect of compromised nodes

It is important to note that the expensive node initialization makes this consensus protocol unsuitable for highly dynamic environments where nodes are created and destroyed upon the system's load. For example, dynamic cloud environments vs. physical machines in a data center.

8.5.2. Security considerations

In addition, when nodes are compromised, unlike in PoW, the compromised node's capability to generate proofs stays constant. Therefore, for an attacker to halt the network's operations, it would require obtaining control of at least 51% of the nodes. Table 5 summarizes the effects that compromised nodes would have on the network.

It is important to note that PoSpace is more resistant to compromised nodes for smaller clusters. For the control plane to be disrupted, a cluster with ten nodes would require at least six to be compromised. Most other considerations are identical to PoW.

8.5.3. Lightweight clients

It is possible to implement light nodes in PoSpace. For light nodes, a full node can create a smaller proof that can convince the light node that the weight of a chain is close to some value. This is called proof of weight. Naively, the light node could download every block in the chain and all the necessary proofs and verify them. However, this would require a lot of bandwidth and CPU.

Conceptually similar to the mechanism used by Flyweight clients [66], a light node can validate through a hashing mechanism that the blockchain in the full node has been fully validated and is the longest consensed chain across the network. This enables the lightweight node to consume the system's state database without participating in the consensus process.

8.6. Proof of Authority

Proof of Authority (PoA) is a reputation-based consensus algorithm that introduces a practical and efficient solution for blockchain networks. The PoA consensus algorithm leverages the value of identities, meaning block validators are arbitrarily selected as trustworthy entities. The weight of the participants in the consensus algorithm is backed by trust.

The identity of a node consists of an asymmetrical key pair. The keys of a node must be generated before the node can be added to the network, as the public key would be part of the resource. When a node is created, it initializes the key pair and stores it in a secure element, for example, a TPM module.

Trusted Platform Module (TPM, also known as ISO/IEC 11889) is an international standard for a secure cryptoprocessor, a dedicated microcontroller designed to secure hardware through integrated cryptographic keys.

8.6.1. Adding a node to the network

A node connects to the P2P network during the setup process without being part of the cluster by authenticating with a pre-created staging key. This key allows the node to download the blockchain from the network. Once the node has a copy of the blockchain, the admin can request from the node the transaction required to add it to the cluster and submit the transaction by providing the locking script. Figure 37 shows an example of a node added to the network, including its public key hash. This enables the node to update any of its metadata. The node uses the staging key to sign the transaction.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest cluster transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "Node",
              "metadata": {
                  "name": "new-node",
```

```
                    "pubKey": "<node_public_key>",
                    "labels": {
                       "name": "offline"
                    }
                }
            },
            "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

Figure 37 Adding node with a public key.

Once the node has been added to the cluster, all other nodes update their peer list with the node's name and the public key that will later be used to validate blocks from the node.

8.6.2. Security concerns

If the staging key is compromised, the attacker can create as many nodes as they want and block the consensus algorithm. The following are possible ways to mitigate the risk of an attacker getting hold of the keys:

1. Create a staging key per node and make the keys single-time use. This process would limit the number of nodes the attacker can create to the number of keys they can get hold of.
2. Limit the rate of nodes that can be added to the cluster.
3. In on-demand environments, it is possible to pre-create all nodes and turn them off. This can also reduce the time to synchronize the blockchain but using de-duplication technology and cloning disk containing the most recent blockchain.
4. Reduce or temporarily eliminate the weight of the new nodes in the consensus algorithm. For example, nodes created in the last 24 hours cannot participate in the consensus algorithm. This introduces other possible

attacks for small clusters where only a portion of the cluster is old enough to join in the consensus, confining all consensus to a small list of nodes.

8.6.3. Consensus algorithm

Nodes on the network compete to produce the next block by signing a proposed block header and broadcasting the signature. The node will enter a contest to be selected to submit the next block.

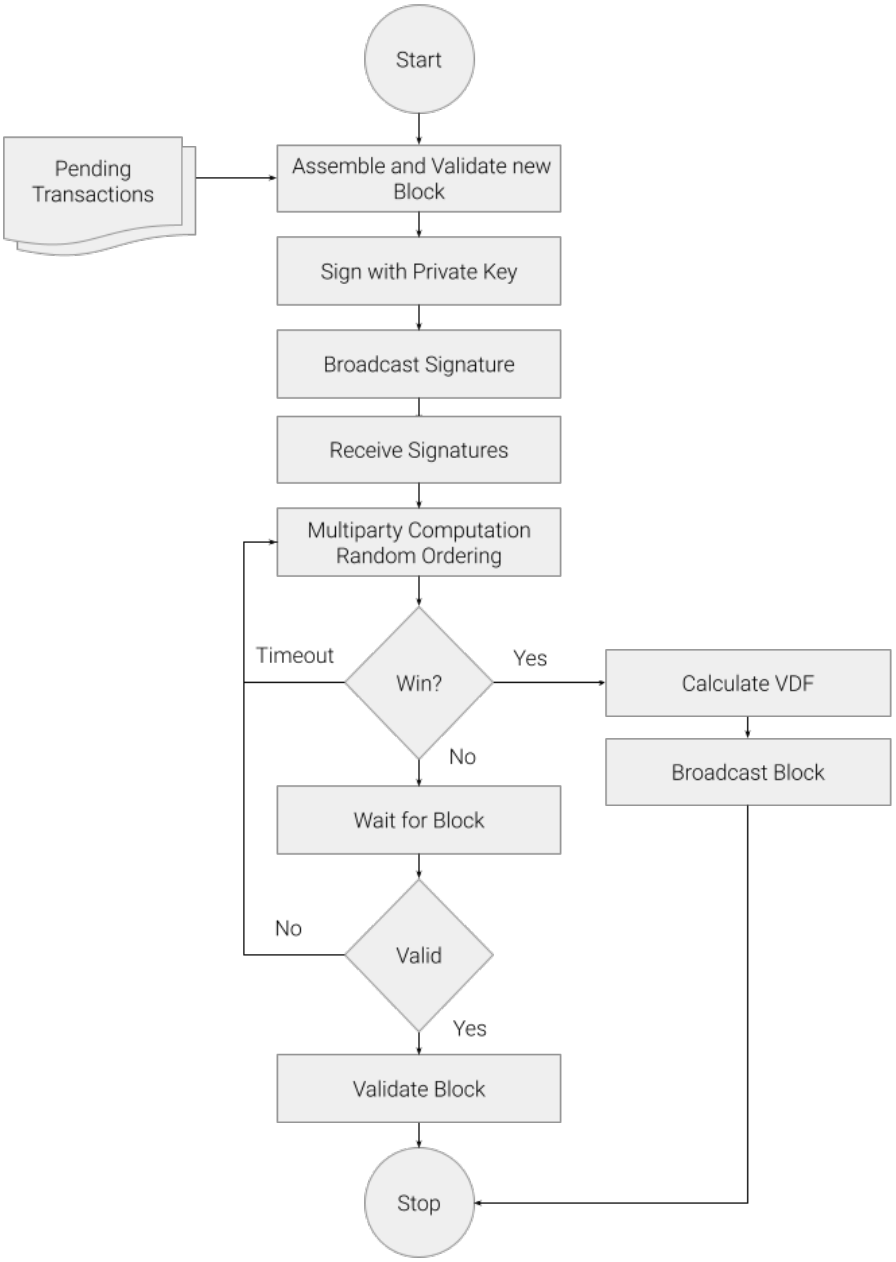

Figure 38 Proof of Authority consensus

The winner will be selected using a collusion-resistant Multiparty Computation Random Ordering algorithm [67]. Once all the nodes receive the results of the random ordering, the first one is to propose the node. If the first one fails to produce a valid block, then the second of the list gets a chance. The process continues until one of the nodes has a valid block. The Verifiable Delay Function is added to the sequence of producing the block to ensure blocks are generated at the predefined cadence.

Other possible mechanisms to ensure a node's fair selection include using Verifiable Random Function [68] (VRF). VRFs require all nodes to share a secret key that can be further randomized with the last block's hash. Seeding the key with the block's hash reduces the possibility of a node pre-calculating which future nodes will be selected to generate the next block.

The consensus has similar characteristics to the PoSpace consensus mechanism with the difference of being resistant to Out of Network attacks where supplemental compute or storage power enhances the capabilities of the adversary to produce blocks.

The significant difference with PoA consensus is that adding nodes to the network needs to be orchestrated. The nodes cannot be added to the block before they are created, and the key pair is registered with the cluster.

Alternatively, a root of trust can be created such that nodes can provide a certificate chain that automatically allows them to become part of the network if the node entries are pre-created in the cluster. The certificate root to generate those certificates becomes a single point of failure for the cluster.

8.6.4. Security Considerations

To further eliminate the possibility of forging blocks that produce a hash that can pre-select the winning node, it is necessary to add the restriction that a node cannot be selected more than once for several blocks proportional to the size of the network. In addition, other heuristics can be added to the algorithm to ensure that a percentage of all the pending transactions must be added to the block for the other nodes to consider the block valid.

All honest nodes will respond, and the computation can be complete. If any nodes do not respond in a specified time, a new round will be started, excluding the nodes that did not reply in the first round. Every time a node fails to respond, it will be blocked for a determined period, for example, 100 blocks.

8.7. Proof of Stake

Proof of Stake (PoS) protocols are a class of consensus mechanisms for blockchains that work by selecting validators in proportion to their quantity of holdings in the associated cryptocurrency. This consensus mechanism is particularly interesting for scenarios where the nodes are managed by parties that do not necessarily trust each by structuring financial compensation in a way that makes an attack less advantageous. If a node

misbehaves (offline, attacks the network, deflects from the protocol) in the network, the process of implementing the penalty is called slashing.

For this research, we will focus on Bitcoin as the financial instrument used in the staking protocols. The cluster blockchain essentially becomes a layer two operation relying on the Bitcoin network to secure the funds at stake required to participate in the consensus algorithm [69].

To orchestrate the financial compensation in the following example, we used the Pay-to-script (P2SH) capability of the Bitcoin network. A P2SH hash allows transactions to be sent to a script hash instead of a public key hash. To spend the amount sent via P2SH, the recipient must provide a script matching the script hash and corresponding signatures making the script evaluated return OP_TRUE. One of the characteristics of P2SH is that the script only needs to be made public when unlocking the funds.

8.7.1 Adding nodes to the network

To enable the protocol, every node needs to be identifiable by a node's public key. The public key uniquely identifies the specific node and is usually presented as a hexadecimal encoding. As in PoA, nodes generate a private root key when first initialized and stored in a secure element or TPM module.

Nodes must submit a Bitcoin on-chain staking transaction that must be verifiable by every other node. The transaction contains the hash signature of a script that requires at least half of the node's signature to unlock the funds and locking period. Because only half of the nodes are required to sign the slashing penalty, the consensus mechanism can enforce it even if some nodes are offline.

In addition, the node can recover the funds without any intervention from the cluster after a lockout period. The time specified on the script is the staking period during which the funds are not recoverable by the node. Figure 39 shows a stalking transaction script with a 5d lockout period.

```
OP_IF
  <N/2>
    <hash_pubk_node_1>
    <hash_pubk_node_2>
…
    <hash_pubk_node_N>
  <N> OP_CHECKMULTISIG
OP_ELSE
  5d
  OP_CHECKSEQUENCEVERIFY
  OP_DROP
  OP_DUP
```

```
  OP_HASH160
  <hash_public_key_from_stalking_node>
  OP_EQUALVERIFY
  OP_CHECKSIG
OP_ENDIF
```

Figure 39 Staking transaction script

The consensus involves two phases:

1. Multiparty computation generates the ordered list of nodes selected to mint the next block. Alternatively, a Verifiable Random Function can be used as stated in Proof of Authority.
2. Node production and validation.

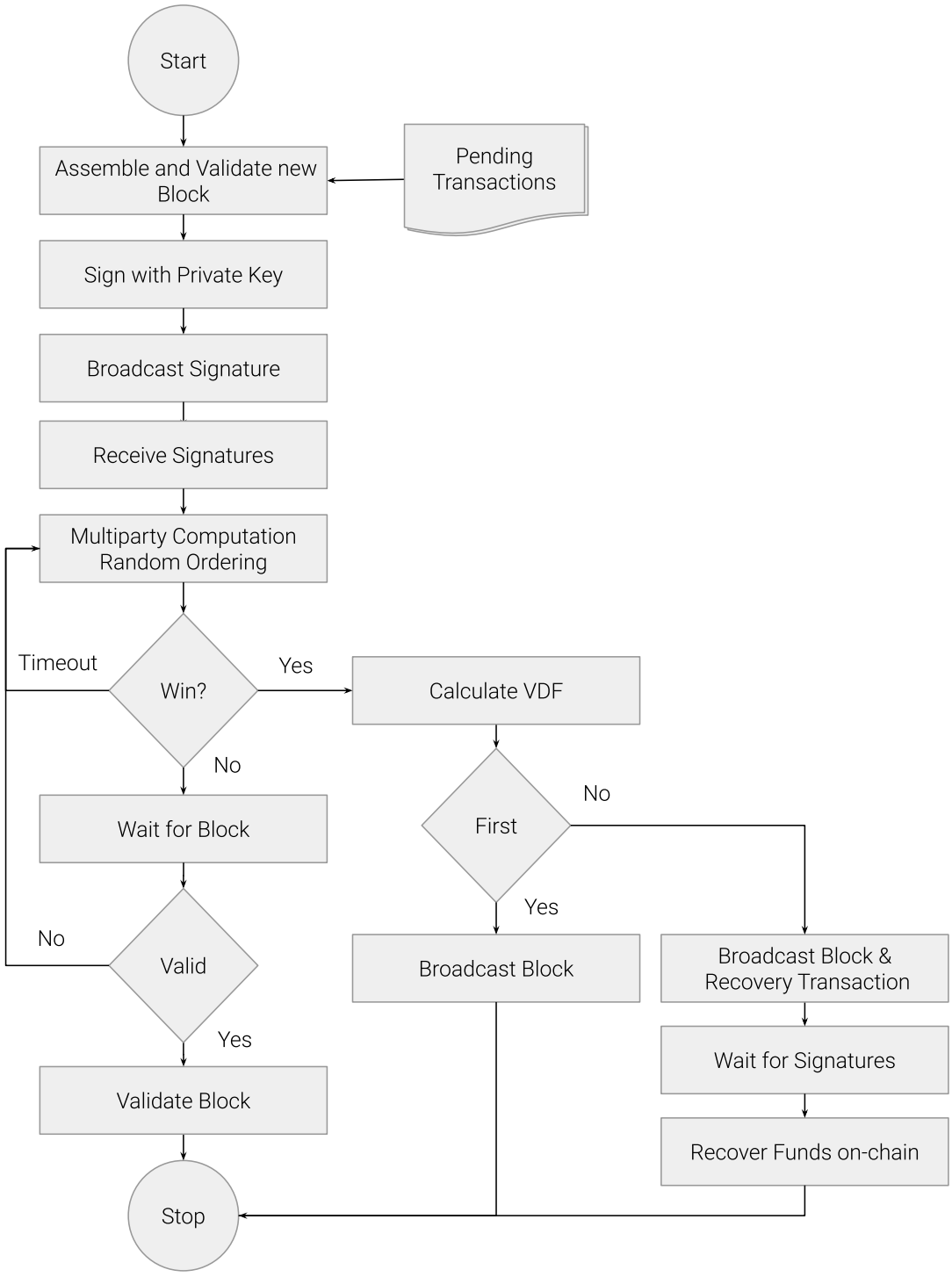

Figure 40 Proof of Stake

8.7.2 Consensus Algorithm

The first phase of the consensus is identical to PoA, where a collusion-resistant multiparty computation is used to ensure a fair selection of candidates to generate the next block. Unlike in the cryptocurrency PoS protocols, all nodes have the same weight in the algorithm. Future research will evaluate the possibility of separating the functions

into minting nodes and validators. After the node is selected, an additional penalty is calculated if the selected node does not produce the next valid block. All candidates must produce the block within a specified timeout. If the node fails to deliver the block, the next node on the multiparty computation is expected to create the next block. This time, the node will also produce a transaction where all nodes that participated in the multi-party computations, minus the nodes that failed to create the block, are compensated. When a node receives a block and a signature, it verifies that it has not received another block already if it has not, the node signs the transaction and sends it to the producer.

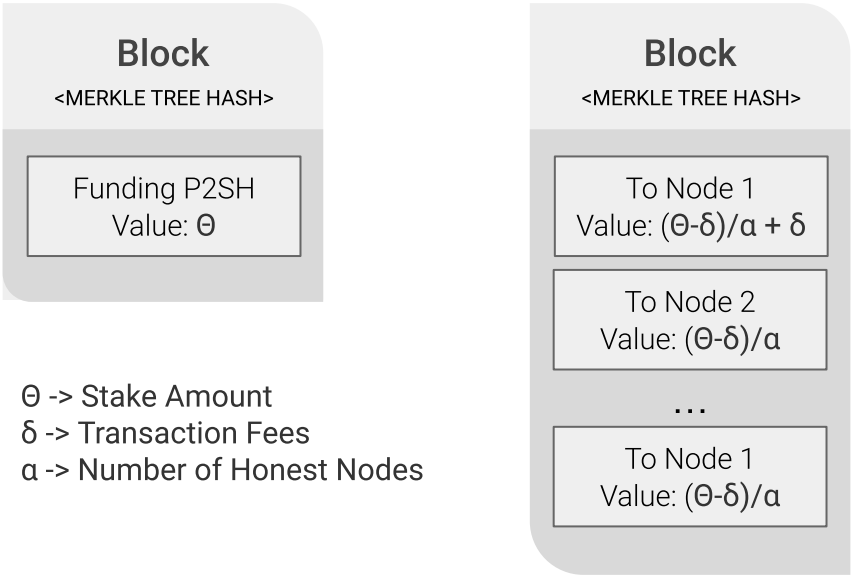

Figure 41 Slashing transaction

Figure 41and Figure 42 are examples of a Bitcoin transaction to be signed by nodes to be compensated for the delay in the consensus.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the staking transaction>",
          "vout": 0,
           "scriptSig": "
<ECDSA Tx Signature Node 1>
<ECDSA Tx Signature Node 2>
```

```
<ECDSA Tx Signature Node 3>
...
<ECDSA Tx Signature Node N>
OP_IF
  <N/2>
    <hash_pubk_node_1>
    <hash_pubk_node_2>
…
    <hash_pubk_node_N>
  <N> OP_CHECKMULTISIG
OP_ELSE
  5d
  OP_CHECKSEQUENCEVERIFY
  OP_DROP
  OP_DUP
  OP_HASH160
  <hash_public_key_from_stalking_node>
  OP_EQUALVERIFY
  OP_CHECKSIG
OP_ENDIF",
           "scriptPubKey": "
OP_HASH160
<hash_public_key>
OP_DUP"
       }
  ],
  "vout": [
      {
          "value": <(stack_amount-fees)/N + fees>,
          "vout": 0,
           "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_block_producing_node>
OP_EQUALVERIFY
OP_CHECKSIG",
      },
       {
```

```
          "value": <(stack_amount-fees)/N>,
          "vout": 1,
           "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_node_1>
OP_EQUALVERIFY
OP_CHECKSIG",
      },

...

       {
          "value": <(stack_amount-fees)/N>,
          "vout": <N>,
           "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_node_N>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

Figure 42 Redistribution transaction

It is essential to notice that in this transaction, the redeem script contains the locking script itself instead of the hash, and the preceding signatures are needed to unlock it. While this would be an expensive operation, it should only happen when nodes fail to perform what they committed to do as part of the consensus.

It is recommended to use Taproot further to increase the performance and privacy of the mechanism. Taproot enables the users to break down the unlocking script into multiple scripts and only reveal the one used to unlock the staked amount. The mechanism relies on Merkle trees. The hash corresponding to every script is added to the Merkle tree, and the tree's root is included in the transaction as part of the unlocking script. To unlock the transaction, the user only needs to provide one of the scripts, the hash and the Merkle tree instead of all the scripts, thus reducing the transaction size.

When a stake distribution event occurs, the node processing the transaction is responsible for paying the transaction fees reimbursed as part of the transaction. The node payment is calculated using the following formula:

$$ProcessorNodeAmount = \frac{StackedAmount - Fees}{NumberParticipatingNodes - NumberFailingNodes} + Fees$$

While the rest of the nodes have the following amount:

$$ProcessorNodeAmount = \frac{StackedAmount - Fees}{NumberParticipatingNodes - NumberFailingNodes}$$

8.7.3. Security Considerations

As in other consensus algorithms, DoS attacks require taking control over most nodes in the network. In contrast, attacks with less than the majority of nodes will be discouraged by losing the capital involved. Even in the case of launching a successful attack, the only thing the attacker can do is disrupt operations. In any case, attackers cannot change the contents of the blockchain and cannot forge blocks.

While beyond the scope of this research, the usage of blockchain platforms that support Turing's complete smart contracts can provide further sophistication to the economic incentive structure and management of funds. A Decentralized Autonomous Organization (DAO) implemented by a smart contract can be used to provide coordination between parties in an automated and trustless environment.

PART III: CONCLUSIONS AND FUTURE RESEARCH

CHAPTER 9. CONCLUSIONS

The proposed architecture provides the foundation for a fully distributed configuration management system that stores the global configuration in a blockchain structure and is distributed across all the nodes in the network. This architecture solution offers improved network-partitioning resistance and availability

The system is available, providing a node is accessible to the user. However, the intent-record consistency is compromised and replaced with casual consistency. In essence, a user querying a different node that received the change might obtain a response that does not include the most recent change, that is, until that change is broadcast through the network and adopted in a block that is part of the longest computed chain. This scenario, we believe, is an acceptable compromise toward autonomy and availability of the system.

The benefits of this decentralized architecture can be summarized as follows:

- Reduced management costs for small and medium deployments.
- Cryptographic proofs replace access control.
- Flexible policies based on a safe scripting language.
- An immutable record of all operations.
- Increased system availability.
- Partition resistance.
- Elimination of central point of failure.

The disadvantages of the proposed system include:

- The increased overhead of computing and storage requirements for each node.
- Casual consistency might make it harder to predict the actual state of the system.
- The Increased complexity of key management and lifecycle.
- The complexity of encoding usage policies as a script.
- Recovery of a security breach is more challenging to contain due to a lack of a single point of control.

9.1. Reduced management Costs

Management costs are categorized as Operational and Capital costs. Operational costs represent the day-to-day costs of deploying and running the system. Capital costs represent the Hardware and Software initial required to run the system. When running in a cloud environment, all HW costs can be re-categorized as Operational costs. Table 6 summarizes the cost differences for the different evaluated approaches.

	Operational	Capital
Centralized	Requires deployment of the controller node in addition to the worker nodes. Requires securing access control mechanism and typically set up zero trust security environment.	Additional controller nodes and distributed storage management systems (Etcd, MySQL) Configuration is centralized and replicated per storage management node. Requires separate logging infrastructure to store operational history and events.
Proof of Work	Requires management of hashing hardware. Additional cost to run PoW algorithm across all worker nodes.	Additional cost of hashing hardware. Configuration replicated across all nodes
Proof of Storage	Requires management of storage devices.	Additional cost of storage devices. Configuration replicated across all nodes.
Proof of Stake	Requires management and custody of staked assets.	Configuration replicated across all nodes.
Proof of Authority	Requires security and management of access control keys.	Configuration replicated across all nodes.

Table 6 Summary of operational and capital costs

For example, a system comprised of 100 worker nodes with 1600 vCPUs and 6000 Gb of RAM would require the investment captured in Table 7. The investment evaluation uses the price calculator of a public cloud provider to calculate the hardware costs and estimates a cost of 5 engineers for a centralized approach vs 3 for the proposed decentralized solutions. Notice as the size of the cluster increases the number of engineers increases at a slower rate, where at some point the cost of running a centralized system becomes cheaper.

	Centralized	PoW	PoSpace	PoStake	PoA
Controller 3x	$600				
Database 3x	$600				
Node 100x (16 vCPUS & 60Gb)	$38000	$38000	$38000	$38000	$38000
Node 100x with GPU (Nvidia T4)		$18000			
Storage 100x (1TB)			$10000		
Logging Node 3x	$600				
Deployment and Maintenance ($150K per engineer)	$60000	$36000	$36000	$36000	$36000
	$99800	$92000	$84000	$74000	$74000

Table 7 Cost comparison

For larger systems, there are other variables that will determine the management cost which would determine the total cost of ownership of the system. Examples of these variables are:

- Geographical distribution of the cluster. Highly dispersed nodes would increase the complexity of the deployment and cost of replicating controller nodes.
- The number of users consuming resources.
- The number of different applications running in the cluster.

9.2. Access Control through cryptographic proofs

In all centralized systems evaluated, access control is determined by Role-based Access Control (RBAC). In these systems, access is determined by the user's role which implies a set of operations that can be performed against the system's resources. Figure 43 is an example of Role that allows access to the API to access "configmaps".

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: development-configmap-manager
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  resourceNames: ["development-configmap"]
  verbs: ["update", "get"]
```

Figure 43 RBAC Security

In effect, RBAC access controls provide a layer of security between the user and the system acting as a gatekeeper to the system resources. Any compromises to this layer will render all access control ineffective as there is no implicit access control at the protocol layer or declarative model.

In the proposed model, the access control policies are encoded and publicly visible on the blockchain. Access control is built-in at the protocol layer and each node in the network cryptographically verifies any transaction. Distributed auditability makes attacks impossible as long as the nodes follow the protocol, and in any case, any violation can be independently verified and detected. In addition, a critical difference between RBAC and the proposed system is the script system that encodes complex policy logic, including cryptographically signed events, time controls, and multi-signature controls. Table 8 shows a high-level comparison of the capabilities between RBAC and Locking Scripts.

	RBAC	Locking Script
Groups	Yes	Yes, via Multisig
Roles	Yes	Possible via external input
Read/Write	Yes	Public Read, Write via Locking Script

Timeouts	Possible through webhooks	Yes
External inputs	Possible through webhooks	Yes, via Oracles
Cryptographic verification	No	Yes
Multi-user access control	No	Yes, via Multisig
Public Auditability	No	Yes
Historical traceability	External Logs	Yes
User controlled security	No	Yes, via custom scripts

Table 8 RBAC vs Locking Scripts

9.3. Blockchain Security

Minting an additional block to the blockchain is perhaps the most critical operation to meet the desired consistency and performance requirements. In future research, we analyzed different algorithms that can potentially be used to ensure that blocks are minted, validated, and added to the blockchain throughout the network while minimizing the amount of trust required. In essence, these algorithms enable the capability to achieve consensus on which blocks to add to the chain based on rules that ensure fairness and security for all participants.

To evaluate the suitability of the different algorithms for internet applications, we compare the different researched algorithms using three properties:

1. Overhead: The capital and operational cost of running the consensus mechanism.
2. Trust setup: The amount of required trust pre-established before participation in the consensus mechanism.
3. Setup speed: The time required to add a node to the network.

	Overhead	Trust Setup	Setup Speed
Proof Of Work	High cost of Operation	Low, protected by Hash Power	Instant
Proof Of Space	High Capital Cost	Low, protected by Proof of Space reserved	Slow, depending on storage size
Proof of Stake	High Capital Reserves	Low, protected by capital reserve	Medium, wait for epoch
Proof of Authority		High, pre-established setup	Instant

Table 9 Consensus algorithm comparison

Table 9 summarizes the properties of the different researched algorithms. Based on this analysis we can infer its suitability for the following different scenarios:

1. Large Internet application: Applications serving users across the globe or multiple regions with variable load demands.
2. Enterprise Cluster: Cluster shared by multiple applications with variable demand.
3. Shared multi-organization: Cluster shared across multiple organizations with multiple applications and variable demand.
4. Development Cluster: Cluster used by development organizations for testing and staging purposes.

The evaluated algorithms are evaluated against these scenarios both On-Cloud and On-Prem in Table 10 based on the properties summarized in Table 9. Consensus algorithms with high setup costs are less suitable for On-Cloud scenarios due to the on-demand nature of these environments. Conversely, those algorithms are well suited for On-Prem environments where the infrastructure is already in place independent of its utilization.

	PoW	PoSpace	PoStake	PoA
Large Internet application On-Cloud	Yes	No	No	Yes
Large Internet application On-Prem	Yes	Yes	No	Yes
On-prem enterprise cluster	Yes	Yes	No	Yes

Multi-cloud enterprise cluster	Yes	No	No	Yes
Shared multi-organization On-Cloud	Yes	No	Yes	No
Shared multi-organization On-Prem	Yes	Yes	Yes	No
Development Cluster	No	No	No	Yes

Table 10 Consensus algorithm suitability

Additionally, consensus algorithms that require pre-establish trust are well suited for highly coordinated organizations that can centralize the cluster operations. For organizations or multiple organizations without centralized trust, Proof-of-Stake algorithms prompt cooperation encouraging good behavior through monetary penalties.

CHAPTER 10. FUTURE RESEARCH

10.1. ZK-SNARKS

“Zero-knowledge” (ZK) proofs allow one participant to prove a verifier that a statement is true, without revealing any information beyond the validity of the statement itself.

ZK proofs can be used to limit the amount of information that the network of nodes needs to know in order to successfully coordinate and distribute applications across the cluster. Scheduling algorithms can operate over range values and the decryption key is only made available to qualified secured nodes.

10.2. DAO Cluster Governance

A decentralized autonomous organization offers cluster administrators a model for the collective management of the cluster resources. DAOs differ from traditional organizations managed by boards and committees enabling a decentralized model for sharing resources across clusters and organizations.

10.3. Confidential computing

Confidential computing uses hardware-integrated solutions to provide a level of assurance of data integrity, data confidentiality, and code integrity. Organizations can run sensitive applications and data on untrusted infrastructure, public clouds, and all other hosted environments. Together with the combination of the rest of the technologies covered in this research, it provides the foundation for the development of a Peer-to-Peer public cloud service provider.

BIBLIOGRAPHY

REFERENCES

[1] A. Berenberg and B. Calder, “Deployment Archetypes for Cloud Applications,” *ACM Comput. Surv.*, vol. 55, no. 3, p. 61:1-61:48, Feb. 2022, doi: 10.1145/3498336.

[2] H. Jamous, “A Reference Architecture for Building Highly Available and Scalable Cloud Application”.

[3] S. S. Gill and R. Buyya, “A Taxonomy and Future Directions for Sustainable Cloud Computing: 360 Degree View.” arXiv, Jul. 09, 2018. Accessed: May 17, 2022. [Online]. Available: http://arxiv.org/abs/1712.02899

[4] A. Gunka, S. Seycek, and H. Kühn, “Moving an application to the cloud: an evolutionary approach,” in *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*, New York, NY, USA, Apr. 2013, pp. 35–42. doi: 10.1145/2462326.2462334.

[5] B. Power, “Digital Transformation Through SaaS Multiclouds,” *IEEE Cloud Computing*, vol. 5, no. 3, pp. 27–30, May 2018, doi: 10.1109/MCC.2018.032591613.

[6] L. Mostarda, S. Marinovic, and N. Dulay, “Distributed Orchestration of Pervasive Services,” in *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, Apr. 2010, pp. 166–173. doi: 10.1109/AINA.2010.100.

[7] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge Computing: Vision and Challenges,” *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, Oct. 2016, doi: 10.1109/JIOT.2016.2579198.

[8] H. Chang, A. Hari, S. Mukherjee, and T. V. Lakshman, “Bringing the cloud to the edge,” in *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2014, pp. 346–351.

[9] D. Bernstein, “Cloud foundry aims to become the OpenStack of PaaS,” *IEEE Cloud Computing*, vol. 1, no. 2, pp. 57–60, 2014.

[10] A. Lomov, “OpenShift and Cloud Foundry PaaS:,” p. 10.

[11] B. Hindman *et al.*, “Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center,” p. 14.

[12] N. Naik, “Building a virtual system of systems using docker swarm in multiple clouds,” in *2016 IEEE International Symposium on Systems Engineering (ISSE)*, Oct. 2016, pp. 1–3. doi: 10.1109/SysEng.2016.7753148.

[13] E. A. Brewer, "Kubernetes and the path to cloud native," in *Proceedings of the sixth ACM symposium on cloud computing*, 2015, pp. 167–167.

[14] "Using the Cloud to build multi-region architecture." https://europeclouds.com/blog/using-the-cloud-to-build-multi-region-architecture (accessed May 17, 2022).

[15] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the Tenth European Conference on Computer Systems*, New York, NY, USA, Apr. 2015, pp. 1–17. doi: 10.1145/2741948.2741964.

[16] H. Qu, O. Mashayekhi, D. Terei, and P. Levis, "Canary: A Scheduling Architecture for High Performance Cloud Computing," p. 13.

[17] G. Sayfan, *Mastering kubernetes*. Packt Publishing Ltd, 2017.

[18] C. Pahl and P. Jamshidi, "Software Architecture for the Cloud – A Roadmap Towards Control-Theoretic, Model-Based Cloud Architecture," in *Software Architecture*, Cham, 2015, pp. 212–220. doi: 10.1007/978-3-319-23727-5_17.

[19] P. Alemany, R. Vilalta, R. Muñoz, R. Casellas, and R. Maríinez, "Peer-to-Peer Blockchain-based NFV Service Platform for End-to-End Network Slice Orchestration Across Multiple NFVI Domains," in *2020 IEEE 3rd 5G World Forum (5GWF)*, Sep. 2020, pp. 151–156. doi: 10.1109/5GWF49715.2020.9221311.

[20] S. Gilbert and N. Lynch, "Perspectives on the CAP Theorem," *Computer*, vol. 45, no. 2, pp. 30–36, Feb. 2012, doi: 10.1109/MC.2011.389.

[21] M. K. Gokhroo, M. C. Govil, and E. S. Pilli, "Detecting and mitigating faults in cloud computing environment," in *2017 3rd International Conference on Computational Intelligence Communication Technology (CICT)*, Feb. 2017, pp. 1–9. doi: 10.1109/CIACT.2017.7977362.

[22] H. C. Lim, S. Babu, J. S. Chase, and S. S. Parekh, "Automated control in cloud computing: challenges and opportunities," in *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, New York, NY, USA, Jun. 2009, pp. 13–18. doi: 10.1145/1555271.1555275.

[23] B. Yang, F. Tan, Y.-S. Dai, and S. Guo, "Performance Evaluation of Cloud Service Considering Fault Recovery," in *Cloud Computing*, Berlin, Heidelberg, 2009, pp. 571–576. doi: 10.1007/978-3-642-10665-1_54.

[24] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001.

[25] Y. Wang, Z. Wang, Y. Chai, and X. Wang, "Rethink the Linearizability Constraints of Raft for Distributed Key-Value Stores," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2021, pp. 1877–1882.

[26] “Failover Pattern with a Self-Healing Mechanism for High Availability Cloud Solutions | IEEE Conference Publication | IEEE Xplore.” https://ieeexplore.ieee.org/abstract/document/6820969/ (accessed May 17, 2022).

[27] “Intent-based cloud service management | IEEE Conference Publication | IEEE Xplore.” https://ieeexplore.ieee.org/abstract/document/8401600/ (accessed May 18, 2022).

[28] “A Comprehensive Study of ‘etcd’—An Open-Source Distributed Key-Value Store with Relevant Distributed Databases | SpringerLink.” https://link.springer.com/chapter/10.1007/978-981-19-0284-0_35 (accessed May 18, 2022).

[29] “An Analysis of Quorum-based Abstractions | Proceedings of the 2018 Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed systems.” https://dl.acm.org/doi/abs/10.1145/3231104.3231957 (accessed May 18, 2022).

[30] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “Conflict-Free Replicated Data Types,” in *Stabilization, Safety, and Security of Distributed Systems*, vol. 6976, X. Défago, F. Petit, and V. Villain, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400. doi: 10.1007/978-3-642-24550-3_29.

[31] “Decentralized Kubernetes Federation Control Plane | IEEE Conference Publication | IEEE Xplore.” https://ieeexplore.ieee.org/abstract/document/9302768 (accessed May 18, 2022).

[32] D. Lindsay, G. Yeung, Y. Elkhatib, and P. Garraghan, “An Empirical Study of Inter-cluster Resource Orchestration within Federated Cloud Clusters,” in *2021 IEEE International Conference on Joint Cloud Computing (JCC)*, Aug. 2021, pp. 44–50. doi: 10.1109/JCC53141.2021.00019.

[33] M. Savi *et al.*, *A Blockchain-based Brokerage Platform for Fog Computing Resource Federation*. 2020. doi: 10.1109/ICIN48450.2020.9059337.

[34] C. Melo, J. Dantas, P. Pereira, and P. Maciel, “Distributed application provisioning over Ethereum-based private and permissioned blockchain: availability modeling, capacity, and costs planning,” *J Supercomput*, vol. 77, no. 9, pp. 9615–9641, Sep. 2021, doi: 10.1007/s11227-020-03617-z.

[35] D. E. Sarmiento, A. Lèbre, L. Nussbaum, and A. Chari, “Decentralized SDN Control Plane for a Distributed Cloud-Edge Infrastructure: A Survey,” *IEEE Communications Surveys & Tutorials*, 2021, doi: 10.1109/COMST.2021.3050297.

[36] C. Felix, H. Garg, and S. Dikaleh, “Kubernetes security and access management: a workshop exploring security & access features in Kubernetes,” in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, USA, Nov. 2019, pp. 395–396.

[37] B. Rochwerger *et al.*, “The Reservoir model and architecture for open federated cloud computing,” *IBM Journal of Research and Development*, vol. 53, no. 4, p. 4:1-4:11, Jul. 2009, doi: 10.1147/JRD.2009.5429058.

[38] C. A. Lee, "Cloud Federation Management and Beyond: Requirements, Relevant Standards, and Gaps," *IEEE Cloud Computing*, vol. 3, no. 1, pp. 42–49, Jan. 2016, doi: 10.1109/MCC.2016.15.

[39] C. Pahl, N. EL Ioini, and S. Helmer, "A Decision Framework for Blockchain Platforms for IoT and Edge Computing:," in *Proceedings of the 3rd International Conference on Internet of Things, Big Data and Security*, Funchal, Madeira, Portugal, 2018, pp. 105–113. doi: 10.5220/0006688601050113.

[40] O. Novo, "Blockchain Meets IoT: An Architecture for Scalable Access Management in IoT," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1184–1195, Apr. 2018, doi: 10.1109/JIOT.2018.2812239.

[41] "A Proof-of-Authority Blockchain Based Distributed Control System for Islanded Microgrids | IEEE Journals & Magazine | IEEE Xplore." https://ieeexplore.ieee.org/document/9681332 (accessed Jun. 06, 2022).

[42] "Ethereum Whitepaper," *ethereum.org*. https://ethereum.org (accessed Jun. 06, 2022).

[43] "Cardano is a decentralized public blockchain and cryptocurrency project and is fully open source.," *Cardano*. https://cardano.org/ (accessed Jun. 06, 2022).

[44] A. Yakovenko, "Solana: A new architecture for a high performance blockchain," p. 32.

[45] S. Aggarwal and N. Kumar, "Chapter Sixteen - Hyperledger☆☆Working model.," in *Advances in Computers*, vol. 121, S. Aggarwal, N. Kumar, and P. Raj, Eds. Elsevier, 2021, pp. 323–343. doi: 10.1016/bs.adcom.2020.08.016.

[46] *BIP 114 - Merkelized Abstract Syntax Tree*. Bitcoin, 2022. Accessed: Jun. 07, 2022. [Online]. Available: https://github.com/bitcoin/bips/blob/b1791c24aa163eb6578d0bfaadcf44997484eeaf/bip-0114.mediawiki

[47] I. Weber *et al.*, "On Availability for Blockchain-Based Systems," in *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, Sep. 2017, pp. 64–73. doi: 10.1109/SRDS.2017.15.

[48] J. M. Hellerstein and P. Alvaro, "Keeping CALM: When Distributed Consistency is Easy." arXiv, Jan. 25, 2019. Accessed: May 18, 2022. [Online]. Available: http://arxiv.org/abs/1901.01930

[49] J. C. Yim, H.-K. Yoo, J. Kwak, and S.-M. Kim, "Blockchain and consensus algorithm," *Electronics and telecommunications trends*, vol. 33, no. 1, pp. 45–56, 2018.

[50] F. Hofmann, S. Wurster, E. Ron, and M. Böhmecke-Schwafert, "The immutability concept of blockchains and benefits of early standardization," in *2017 ITU Kaleidoscope: Challenges for a Data-Driven Society (ITU K)*, Nov. 2017, pp. 1–8. doi: 10.23919/ITU-WT.2017.8247004.

[51] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: a survey," *International Journal of Web and Grid Services*, vol. 14, no. 4, pp. 352–375, Jan. 2018, doi: 10.1504/IJWGS.2018.095647.

[52] C. Li, P. Li, D. Zhou, W. Xu, F. Long, and A. Yao, "Scaling Nakamoto Consensus to Thousands of Transactions per Second," arXiv, arXiv:1805.03870, Aug. 2018. doi: 10.48550/arXiv.1805.03870.

[53] Y. Sun, "Commutativity-aware Runtime Verification for Concurrent Programs," Thesis, 2021. Accessed: Jun. 05, 2022. [Online]. Available: https://oaktrust.library.tamu.edu/handle/1969.1/193203

[54] D. Ernst, A. Becker, and S. Tai, "Rapid Canary Assessment Through Proxying and Two-Stage Load Balancing," in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, Mar. 2019, pp. 116–122. doi: 10.1109/ICSA-C.2019.00028.

[55] A. Jeffery, H. Howard, and R. Mortier, "Rearchitecting Kubernetes for the Edge," in *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, New York, NY, USA, Apr. 2021, pp. 7–12. doi: 10.1145/3434770.3459730.

[56] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Kubernetes as an Availability Manager for Microservice Applications." arXiv, Jan. 15, 2019. Accessed: May 18, 2022. [Online]. Available: http://arxiv.org/abs/1901.04946

[57] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, Jul. 2018, pp. 970–973. doi: 10.1109/CLOUD.2018.00148.

[58] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "A Kubernetes controller for managing the availability of elastic microservice based stateful applications," *Journal of Systems and Software*, vol. 175, p. 110924, May 2021, doi: 10.1016/j.jss.2021.110924.

[59] A. Singh, T. "johnny Ngan, P. Druschel, and D. S. Wallach, "Eclipse attacks on overlay networks: Threats and defenses," 2006.

[60] T. C. Aysal, M. E. Yildiz, and A. Scaglione, "Broadcast gossip algorithms," in *2008 IEEE Information Theory Workshop*, May 2008, pp. 343–347. doi: 10.1109/ITW.2008.4578682.

[61] W. Fan, S.-Y. Chang, X. Zhou, and S. Xu, "ConMan: A Connection Manipulation-based Attack Against Bitcoin Networking," in *2021 IEEE Conference on Communications and Network Security (CNS)*, Oct. 2021, pp. 101–109. doi: 10.1109/CNS53000.2021.9705018.

[62] T. C. Aysal, M. E. Yildiz, A. D. Sarwate, and A. Scaglione, "Broadcast Gossip Algorithms for Consensus," *IEEE Transactions on Signal Processing*, vol. 57, no. 7, pp. 2748–2761, Jul. 2009, doi: 10.1109/TSP.2009.2016247.

[63] C. H. Moore and G. C. Leach, “FORTH - A Language for Interactive Computing,” p. 27.

[64] S. Lucas, “The origins of the halting problem,” *Journal of Logical and Algebraic Methods in Programming*, vol. 121, p. 100687, Jun. 2021, doi: 10.1016/j.jlamp.2021.100687.

[65] G. Ateniese, I. Bonacina, A. Faonio, and N. Galesi, “Proofs of Space: When Space Is of the Essence,” in *Security and Cryptography for Networks*, Cham, 2014, pp. 538–557. doi: 10.1007/978-3-319-10879-7_31.

[66] B. Bünz, L. Kiffer, L. Luu, and M. Zamani, “FlyClient: Super-Light Clients for Cryptocurrencies,” in *2020 IEEE Symposium on Security and Privacy (SP)*, May 2020, pp. 928–946. doi: 10.1109/SP40000.2020.00049.

[67] D. Bogdanov, S. Laur, and R. Talviste, “A Practical Analysis of Oblivious Sorting Algorithms for Secure Multi-party Computation,” in *Secure IT Systems*, Cham, 2014, pp. 59–74. doi: 10.1007/978-3-319-11599-3_4.

[68] Y. Dodis and A. Yampolskiy, “A Verifiable Random Function with Short Proofs and Keys,” in *Public Key Cryptography - PKC 2005*, Berlin, Heidelberg, 2005, pp. 416–431. doi: 10.1007/978-3-540-30580-4_28.

[69] E. N. Tas, D. Tse, F. Yu, and S. Kannan, “Babylon: Reusing Bitcoin Mining to Enhance Proof-of-Stake Security,” arXiv, arXiv:2201.07946, Jan. 2022. doi: 10.48550/arXiv.2201.07946.

[70] V. Buterin, “Ethereum white paper,” *GitHub repository*, vol. 1, pp. 22–23, 2013.

[71] S.-H. Chun and B.-S. Choi, “Service models and pricing schemes for cloud computing,” *Cluster Comput*, vol. 17, no. 2, pp. 529–535, Jun. 2014, doi: 10.1007/s10586-013-0296-1.

[72] G. Laatikainen, A. Ojala, and O. Mazhelis, “Cloud Services Pricing Models,” in *Software Business. From Physical Products to Software Services and Solutions*, Berlin, Heidelberg, 2013, pp. 117–129. doi: 10.1007/978-3-642-39336-5_12.

[73] *8x Protocol - Whitepaper*. Helis Network, 2019. Accessed: Jun. 08, 2022. [Online]. Available: https://github.com/helisnetwork/8x-whitepaper

[74] “EIP-721: Non-Fungible Token Standard,” *Ethereum Improvement Proposals*. https://eips.ethereum.org/EIPS/eip-721 (accessed Jun. 08, 2022).

[75] *AztecProtocol/AZTEC*. Aztec Network, 2022. Accessed: Jun. 08, 2022. [Online]. Available: https://github.com/AztecProtocol/AZTEC/blob/7a020f4ced9680f6e4a452fe570671aac0802471/AZTEC.pdf

[76] B. Thompson, “Shopify and the Power of Platforms,” *Stratechery by Ben Thompson*, Jul. 11, 2019. https://stratechery.com/2019/shopify-and-the-power-of-platforms/ (accessed Jun. 08, 2022).

[77] B. Thompson, "Defining Aggregators," *Stratechery by Ben Thompson*, Sep. 26, 2017. https://stratechery.com/2017/defining-aggregators/ (accessed Jun. 08, 2022).

[78] "A Multi-party Protocol for Constructing the Public Parameters of the Pinocchio zk-SNARK | SpringerLink." https://link.springer.com/chapter/10.1007/978-3-662-58820-8_5 (accessed Jun. 08, 2022).

[79] "The Maker Protocol White Paper | Feb 2020." https://makerdao.com/en/whitepaper/https://makerdao.com/en (accessed Jun. 08, 2022).

APPENDIX 1. SERVICE MONETIZATION

BLOCKCHAIN-BASED SERVICE MONETIZATION FOR CLOUD SERVICES

1.1. Introduction

While many cryptocurrencies enable peer-to-peer payments without a trusted party [70], none provide mechanisms to allow payee-initiated or recurring payments using public permissionless blockchains with decentralized validators. It is also interesting to contrast that Software and Infrastructure as a service have come to rely exclusively on business models built on recurring payments and subscriptions [60]. In effect, the world of cloud services has not been able to enjoy the benefits of decentralized finance due to a mismatch of payment and contracting motions.

At the core of the cloud subscription models is the drive to reinvent industries seeking to build personalized relationships with their consumers. A cloud service's success depends on an organization's ability to adapt and enable flexible consumption models that match the needs of its target market [71]. Fundamentally, service pricing needs to quantify the value provided to the consumer accurately. Furthermore, the price of a service is a delicate and ever-changing orchestration of the relationship with the consumer, who constantly re-considers the value provided by the service producer.

For this research, we have classified pricing models into two broad categories: usage-based and capability-based pricing models [72].

- Usage-based pricing models are well suited for services where a consumer can naturally identify a discrete, quantifiable property of a service that represents the value provided. For example, consumers of a data storage service can naturally determine the amount of data consumed as a correlated metric of the value provided.
- Capability-based pricing models are well suited for services where the value provided is not directly proportional to quantifiable metrics associated with the service delivered. The service price is then segmented into desirable features for the different target consumers and either priced individually or bundled.

Capability-based pricing models enable consumers to estimate costs before consumption. In contrast, usage-based pricing models typically require forecasting based on previous consumption and careful evaluation of evolving consumption

patterns. In addition, subscriptions typically involve complex commercial patterns, including term commitment discounts, promotions, prepayments, and sales and channel commissions, among many others.

The current state of the art only scratches the surface of these complex commercial motions. This work outlines an approach to enable commercial-grade subscription models on the Ethereum blockchain. We are publishing this research to outline protocols expressed as Ethereum smart contracts to extend state of the art for the negotiation and commitment phases.

Previous attempts to implement recurring payments smart contracts over the Ethereum blockchain have been abandoned and have not been verified [73]. In addition, our approach provides an explicit separation between the administrative and settlement functions.

The referenced solutions include smart contracts that standardize the interfaces used by consumers and producers to express their commercial relationships. This research does not cover off-chain solutions to enable the execution of payments or offer solutions to custodial issues.

1.2. System Model and Methods

The proposed solution builds on the standard ERC-721 [74] interface, issuing a Non-Fungible Token (NFT) for each subscription. ERC-721 enables consumers to adopt existing NFT management technology, including crypto Wallets and interfaces to manage ownership, transfers, and approvals.

```
interface ISubscription {
    /// @dev This emits when the Subscription Contract is created
    event ContractCreated(
        address indexed _from,
        uint256 indexed subscriptionId,
        uint256 periodLength,
        uint256 periodCount,
        uint256 periodCost,
        bytes data
    );

    /// @dev This emits when the Subscription Contract is signed
    event ContractSigned(
        address indexed _from,
        uint256 indexed _subscriptionId);

    /// @dev This emits when a subscription is renewed
```

```
    event Renewed(
        address indexed _from,
        uint256 subscriptionId,
        uint256 periodLength,
        uint256 periodCount,
        uint256 periodCost
    );

    /// @dev This emits when the state of a Subscription is
changed.
    event StatusChanged(
        address indexed _from,
        uint256 indexed subscriptionId,
        SubscriptionStatus status
    );

    /// @notice Creates a new subscription contract and waits for
the
    ///  consumer to sign it
    /// @param _subscriptionId to be used as NFT tokenId
    /// @param _to Address of the consumer that is receiving the
service
    /// @param _serviceURI The service unique resource identifier
    /// @param _periodLength The length of the subscription period
in
    ///  days
    /// @param _periodCount The period number. If `periodCount` ==
0 the
    ///  subscription is open ended
    /// @param _periodCost The cost per period
    /// @param _data Additional data with no specified format, sent
in
    ///  call to `_to` see ERC721TokenReceiver for reference
    function createSubscriptionContract(
        address _to,
        uint256 _subscriptionId,
        string memory _serviceURI,
        uint256 _periodLength,
        uint256 _periodCount,
```

```
        uint256 _periodCost,
        bytes memory _data
    ) external;

    /// @notice The consumer signs the subscription and the NFT is
    ///  transferred to the `_to` address using SafeTransferFrom
    /// @param _subscriptionId Throws if `_subscriptionId` is not a
valid
    ///  Subscription address or `msg.sender` does not match the
`_to`
    ///  address for this SubscriptionContract
    function signSubscriptionContract(uint256 _subscriptionId)
external;

    /// @notice Returns the status of the subscription
    /// @param _subscriptionId Throws if `_subscriptionId` is not a
valid
    ///  Subscription
    /// @return Status of the subscription unless throwing
    function getSubscriptionStatus(
        uint256 _subscriptionId
    ) external view returns (SubscriptionStatus);

    /// @notice Creates a new subscription contract using the terms
of
    ///  of the original subscription. The periodCount will include
any
    ///  number of periods left on the original subscription. The
    ///  previous
    ///  subscription status will change to RENEWED.
    /// @param _subscriptionId Throws if `_subscriptionId` is not a
valid
    ///  Subscription or is not in valid state
    /// @param _newSubscriptionId to be used as NFT tokenId for the
new
    ///  Subscription
    function renewSubscription(
        uint256 _subscriptionId,
        uint256 _newSubscriptionId) external;
```

```
    /// @notice Set the status of a subscription to `PAUSED`.
Throws if
    ///  subscription cannot be paused
    /// @param _subscriptionId Throws if `_subscriptionId` is not a
valid
    ///  Subscription.
    function pauseSubscription(uint256 _subscriptionId) external;
    /// @notice Set the status of a subscription to `PAUSED`.
Throws if
    ///  subscription cannot be resumed
    /// @param _subscriptionId Throws if `_subscriptionId` is not a
valid
    ///  Subscription.
    function resumeSubscription(uint256 _subscriptionId) external;

    /// @notice Set the status of a subscription to `TERMINATED`.
Throws
    ///  if subscription cannot be terminated
    /// @param _subscriptionId Throws if `_subscriptionId` is not a
valid
    ///  Subscription
    function terminateSubscription(uint256 _subscriptionId)
external;

    /// @notice Set the status of a subscription to `CANCELLED`.
Throws
    ///  if subscription cannot be canceled
    /// @param _subscriptionId Throws if `_subscriptionId` is not a
valid
    ///  Subscription
    function cancelSubscription(uint256 _subscriptionId) external;
}
```

Figure 44 Subscription solidity interface

Every Subscription Contract is identified by a unique ID represented as a non-Fungible token implementing the ERC-721 interface as presented in Figure 44 and available at https://github.com/thinkelastic/subcrypto.

The producer or intermediary creates a subscription. The NFT representing the Subscription will be owned by the address of the entity that created it. Once the subscription contract is instantiated, the state of the subscription is initialized as OFFERED and ready to be signed by the address provided during the creation. Essential to the spirit of NFTs, the token represents consumer authority over the subscription, and until it is signed, that authority is not transferred to the consumer.

The producer waits until the consumer signs the contract. The signing also implies acceptance of the terms and conditions of the service, including any financial responsibility. It is worth pointing out that the financial burden of the contract does not transfer with the change of control of the NFT and stays with the consumer even if the NFT is assigned to another address. Once the contract is signed, the subscription contract NFT is transferred to the consumer, and the subscription status is changed to ACTIVE. Therefore, the subscription term starts the moment the consumer signs the subscription.

As stated before, the consumer can use the ownership of this NFT as an assertion of ownership across other systems. The NFT can be transferred to other accounts, therefore, delegating the rights that the NFT ownership might entitle but not the financial liability incurred by accepting the original offer.

The state machine in Figure 45 represents the valid state transitions of a subscription. Once the subscription is ACTIVE, the consumer might pause the subscription if the contract implementation allows it. Pausing a subscription does interrupt the service delivery, but it typically does not extend the term while the service is paused.

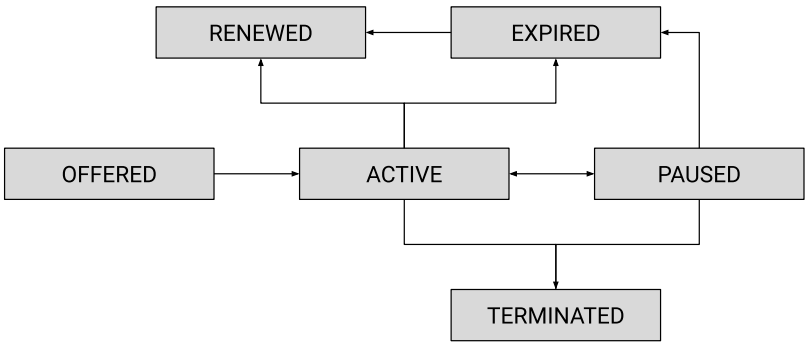

Figure 45 Subscription state machine

Some producers might extend the term length and provide rebates or alternative promotion mechanisms to give customers extra flexibility. In addition, a producer might pause a service, for example, to address a customer not meeting the terms of service. Pausing enables the service to be disabled, allowing the consumer or the producer to restore the service to its original functional state. A resumed subscription would revert to the ACTIVE state.

Function	Caller	Validation	Valid States
`createSubscriptionContract`	Producer	The person calling this function is automatically designed as a producer. The _to parameter designates the address of the consumer.	N/A
`signSubscriptionContract`	Consumer	Only the consumer is allowed to sign the contract.	OFFERED
`getSubscriptionStatus`	Anybody		Any state
`renewSubscription`	Consumer	Only the consumer is allowed to renew the subscription.	EXPIRED, ACTIVE
`pauseSubscription`	Consumer, Producer		ACTIVE
`resumeSubscription`	Consumer, Producer		PAUSED
`terminateSubscription`	Producer	Only the producer can terminate the subscription.	ACTIVE, PAUSED
`cancelSubscription`	Consumer	Only the consumer is allowed to cancel the subscription.	ACTIVE, PAUSED

Table 11 Permissions and valid transitions

Subscriptions that are either Active or Paused can be terminated by either the producer or the consumer. The smart contract implementation separates these two actions into two separate functions. Active or Expired subscriptions can be renewed. Renewals or modifications of a subscription are performed by issuing a new NFT that replaces the original contract. Other than the status of the NFT, the subscription data is immutable.

The status of the subscription is stored as part of the contract state. When invoking getSubscriptionStatus, a check needs to be performed on whether the subscription has expired and updates the internal state accordingly.

To ensure that only the involved parties interact with the contract, the implementation needs to keep track of both the consumer and the producer's identities. Each function would require verification that the subscription is in the right state and that caller is allowed to invoke the function. Table 11 Permissions and valid transitions summarizes the high-level permission logic.

The owner of the token is the designated consumer of the subscription. Once the subscription has been signed, an ERC721 compatible token is transferred to the consumer.

1.3. Results

In this research, we have presented an implementation of a subscription model that demonstrates the capabilities of permissionless blockchains to remove the platform walls that surround modern e-Commerce systems, opening the infrastructure that supports producer-consumer relationships. This translates to the following benefits:

1. Standardized system interface and logic transparency enabling higher productivity and trust across all parties.
2. Blockchain architecture increases automation and productivity because all operations must be programmatic instead of manual. There are also no exceptions to the contract logic because system rules cannot be modified.
3. Transparency, consumers, and producers have equal access to data.
4. Privacy, producers, and consumers can optionally hide their transactions using Zero Proof networks, for example, the AZTEC Protocol [75].
5. Due to the system's openness and transparency, a new ecosystem can arise without the producer's permission.

The most basic form of contractual negotiation is to commit to consuming services using the list price and the default terms offered by the producer. Most subscription options are Monthly, Annual, or Quarterly terms with a limited menu of options for consumers. However, negotiation and contracting needs vary depending on the nature of the service.

Large-scale service producers typically offer subscription packages that rely on simplicity and consistency via single capability pricing or 2 to 3 bundled options for producers providing more than one service. The goal is to eliminate the friction of adoption and maximize the appeal to the maximum number of consumers. These services include cloud storage, streaming, and online gaming services.

Service producers of commoditized services where little differentiation is possible and negligible platform effects tend to focus on pricing targeting customer commitment, including term discounts and signing up for automatic recurring payments and renewals. In some cases, prepayment for several years of the service. These services include internet providers, cell phone providers, VPNs, and security services. To facilitate the pricing structure of these services, focus on capabilities with usage-based overages to target the maximum number of users with attractive pricing while monetizing consumers beyond standard utilization patterns.

Service producers for specialized or domain-specific services' primary concern are to enable target consumers to try, evaluate and ramp up utilization. These services are the most heterogeneous in nature and typically require contractual flexibility and, in most cases, split commissions across parties responsible for the success of the contractual process. Most enterprise software services follow this pattern as the needs of large organizations is distinct enough, making it hard to identify commonalities.

Over the last two decades, we have seen a new model of horizontal and vertical integration in the software distribution model due to the elimination of distribution costs enabled by the internet.

- Platforms aim to gain a competitive advantage by delivering a complete vertical solution for an entire domain. The goal is to leverage the platform effects of providing related domain services by reducing consumption complexity and the operational surface. These services include AWS, Azure, Google Cloud, Adobe Cloud, and Office 365 [76].

- Aggregators aim to gain a competitive advantage by gaining a horizontal monopoly on suppliers, distributors, and consumers/users. Gaining such market exclusivity allows them to provide better value at a reduced cost due to the economies of scale. These services include Netflix, Airbnb, and Uber [77].

Both platforms and aggregators seek to control all aspects of user interaction, including providing payment processing and credit cards to enhance the user experience and ease of use. Nowadays, service producers not part of a Platform or Aggregator are at a disadvantage. At the same time, consumers experience a reduced set of options and restricted functionality as they gravitate toward the ease of use of those platforms.

Essential to the spirit of permissionless blockchains like Ethereum is the ability for any party to produce and consume capabilities like those found on platforms and aggregators as Smart Contracts. These mechanisms, in aggregate, can be reasonably expected to fulfill the lifecycle of commercial contracts like the ones previously discussed.

Most producers rely on channel partners to distribute, sell, and support their products. Although online service producers seek a close feedback loop with consumers, factors including support and financial and regulatory requirements need the specialized capabilities of those partners.

Traditional channel relationships are usually fraught with commercial issues, most of them derived from the lack of transparency of the commercial ledger. Examples are:

1. Exclusivity violations
2. Price discrimination
3. Vertical Non-Price Restraints
4. Territorial and Customer Restrictions
5. Tying
6. Legal embargos
7. Denial of service on moral grounds

1.4. Conclusions

This research has established the benefits of ledger transparency provided by smart contracts on permissionless blockchains. However, most businesses would consider their business information critically confidential. To address this issue, further research will be performed to evaluate Zero-knowledge proof protocols that enable the different parties to prove assertions that meet the transparency requirements without revealing confidential information [78]. For example, using the AZTEC protocol to facilitate private transactions on Ethereum. This allows the logic of transactions to be validated while keeping the values encrypted.

As organizations conduct business utilizing blockchains as transaction ledgers, in aggregate, it can be reasonably expected for the value of those transactions to maintain a stable price relative to the monetary denomination of the rest of their business.

The Maker Protocol enables those businesses to execute stable coin-denominated transactions. In some sense, this sounds ideal; however, it adds the additional complexity of having customers convert their cryptocurrency to a contracted stable coin, thus assuming the volatility risk [79]. For example, a consumer might commit to a transaction that requires a monthly payment of DAI tokens pegged against the dollar at a ratio of $1 = 1 DAI. If the consumer has his assets denominated in ETH, changes in the value of ETH can positively or negatively affect the total contract value for the consumer.

The critical point is that consumers and producers can protect the transaction value of contracts executed through the blockchain utilizing DeFI protocols. Still, the way this is achieved is beyond the scope of this paper.

APPENDIX 2: RESOURCE OPERATIONS

RESOURCE OPERATIONS

2.1. Node

2.1.1. Create Node

To create a node, the user needs to reference the latest cluster transaction as input and a scriptSig value that satisfies the input transaction script.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest cluster transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "Node",
              "metadata": {
                  "name": "new-node",
                   "pubKey": "<node_public_key>",
                   "labels": {
                      "name": "online"
                  }
              }
          },
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
```

```
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.1.2. Update Node

To update a node, the user needs to reference the latest cluster or node transaction as input and a scriptSig value that satisfies the input transaction script. The value field must contain a fully updated Node document.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest node transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "Node",
              "metadata": {
                  "name": "new-node",
                   "pubKey": "<node_public_key>",
                   "labels": {
                      "name": "offline"
                  }
              }
          },
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
```

```
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.1.3. Remove Node

To delete a node, the user needs to reference the latest cluster or node transaction as input and a scriptSig value that satisfies the input transaction script. The value of the transaction must be null.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest node transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": null,
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.2. Namespace

2.2.1. Create Namespace

To create a namespace, the user needs to reference the latest cluster transaction as input and a scriptSig value that satisfies the input transaction script.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest cluster transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "Namespace",
              "metadata": {
                  "name": "development",
                   "labels": {
                      "name": "beta"
                  }
              }
          },
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.2.2. Update Namespace

To update a namespace, the user needs to reference the latest cluster or namespace transaction as input and a scriptSig value that satisfy the input transaction script. The value field must contain a fully updated Namespace document.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest namespace
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "Namespace",
              "metadata": {
                  "name": "development",
                   "labels": {
                      "name": "release-candidate"
                  }
              }
          },
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.2.3. Remove Namespace

To delete a node, the user needs to reference the latest cluster or namespace transaction as input and a scriptSig value that satisfy the input transaction script. The value of the transaction must be null.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest namespace
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": null,
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.3. Pod

2.3.1. Create Pod

To create a pod, the user needs to reference the latest namespace transaction as input and a scriptSig value that satisfies the input transaction script.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest namespace
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "Pod",
              "metadata": {
                  "name": "nginx",
                  "labels": {
                      "name": "development"
                  }
              },
              "spec": {
                  "containers": {
                       “name”: "nginx",
                       “image”: "nginx",
                       “ports”: {
                            "containerPort": [ 80 ]
                       }
                   }
              }
          },
```

```
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.3.2. Update Pod

To update a pod, the user needs to reference the latest namespace or pod transaction as input and a scriptSig value that satisfy the input transaction script. The value field must contain a fully updated Pod document.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest pod transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "Pod",
              "metadata": {
                  "name": "nginx",

                  "labels": {
                      "name": "release-candidate"
                  }
              },
              "spec": {
```

```
                    "containers": {
                        “name”: "nginx",
                        “image”: "nginx",
                        “ports”: {
                            "containerPort": [ 8080 ]
                        }
                    }
                }
            },
            "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
        }
    ]
}
```

2.3.3. Remove Pod

To delete a pod, the user needs to reference the latest namespace or pod transaction as input and a scriptSig value that satisfy the input transaction script. The value of the transaction must be null.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest pod transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": null,
          "scriptPubKey": "
```

```
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.4. Service

2.4.1. Create Service

To create a service, the user needs to reference the latest namespace transaction as input and a scriptSig value that satisfies the input transaction script.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest namespace
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "Service",
              "metadata": {
                  "name": "nginx-service"
              },
              "spec": {
                  "selector": {
                     “name”: “development”
                  },
                  "ports": [
                      {
                            “protocol”: "nginx",
                            “port”: 80,
                            "targetPort": 8080
                      }
                  }
              }
          },
```

```
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.3.2. Update Service

To update a service, the user needs to reference the latest namespace or service transaction as input and a scriptSig value that satisfy the input transaction script. The value field must contain a fully updated Service document.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest service transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "Service",
              "metadata": {
                  "name": "nginx",
              },
              "spec": {
                  "selector": {
                      “name”: “release-candidate”
                  },
                  "ports": [
```

```
                        {
                              “protocol”: "nginx",
                              “port”: 8080,
                              "targetPort": 8080
                        }
                    }
                }
            },
            "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
        }
    ]
}
```

2.4.3. Remove Service

To delete a service, the user needs to reference the latest namespace or service transaction as input and a scriptSig value that satisfy the input transaction script. The value of the transaction must be null.

```
{
    "version": 1,
    "vin": [
        {
            "txid": "<txid from the latest service transaction>",
            "vout": 0,
            "scriptSig": "<ECDSA Tx Signature><PublicKey>"
        }
    ],
    "vout": [
        {
            "value": null,
            "scriptPubKey": "
OP_DUP
```

```
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.5. Deployment

2.5.1. Create Deployment

To create a deployment, the user needs to reference the latest namespace transaction as input and a scriptSig value that satisfies the input transaction script.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest namespace
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "Deployment",
              "metadata": {
                  "name": "nginx-deployment"
              },
              "spec": {
                  "replicas": 3,
                  "selector": {
                     “matchLabels”: {
                         “name”: “development”
                      }
                  },
                  "template": {
                      “metadata”: {
                          “labels”: {
                              “name”: “development”
                          }
                      },
```

```
                        "spec": {
                            "containers": {
                                “name”: "nginx",
                                “image”: "nginx",
                                “ports”: {
                                    "containerPort": [ 80 ]
                                }
                            }
                        }
                    }
                }
            },
            "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
        }
    ]
}
```

2.5.2. Update Deployment

To update a deployment, the user needs to reference the latest namespace or deployment transaction as input and a scriptSig value that satisfy the input transaction script. The value field must contain a fully updated Deployment document.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest deployment
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
```

```
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "Deployment",
              "metadata": {
                  "name": "nginx-deployment"
              },
              "spec": {
                  "replicas": 3,
                  "selector": {
                     “matchLabels”: {
                         “name”: “release-candidate”
                      }
                  },
                  "template": {
                      “metadata”: {
                          “labels”: {
                              “name”: “release-candidate”
                          }
                      },
                      "spec": {
                          "containers": {
                              “name”: "nginx",
                              “image”: "nginx",
                              “ports”: {
                                  "containerPort": [ 8080 ]
                              }
                          }
                      }
                  }
              }
          },
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
```

```
OP_CHECKSIG",
      }
  ]
}
```

2.5.3. Remove Deployment

To delete a deployment, the user needs to reference the latest namespace or deployment transaction as input and a scriptSig value that satisfy the input transaction script. The value of the transaction must be null.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest deployment
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": null,
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.6. ReplicationController

2.6.1. Create ReplicationController

To create a ReplicationController, the user needs to reference the latest namespace transaction as input and a scriptSig value that satisfies the input transaction script.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest namespace
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "ReplicationController",
              "metadata": {
                  "name": "nginx-deployment"
              },
              "spec": {
                  "replicas": 3,
                  "selector": {
                     “matchLabels”: {
                         “name”: “development”
                      }
                  },
                  "template": {
                      “metadata”: {
                          “labels”: {
                              “name”: “development”
                          }
                      },
```

```
                        "spec": {
                            "containers": {
                                “name”: "nginx",
                                “image”: "nginx",
                                “ports”: {
                                    "containerPort": [ 80 ]
                                }
                            }
                        }
                    }
                }
            },
            "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
        }
    ]
}
```

2.6.2. Update ReplicationController

To update a ReplicationController, the user needs to reference the latest namespace or ReplicationController transaction as input and a scriptSig value that satisfy the input transaction script. The value field must contain a fully updated ReplicationController document.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest ReplicationController
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
```

```
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "ReplicationController",
              "metadata": {
                  "name": "nginx-deployment"
              },
              "spec": {
                  "replicas": 3,
                  "selector": {
                     “matchLabels”: {
                         “name”: “release-candidate”
                      }
                  },
                  "template": {
                      “metadata”: {
                          “labels”: {
                              “name”: “release-candidate”
                          }
                      },
                      "spec": {
                          "containers": {
                              “name”: "nginx",
                              “image”: "nginx",
                              “ports”: {
                                  "containerPort": [ 8080 ]
                              }
                          }
                      }
                  }
              },
              "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
```

```
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.6.3. Remove ReplicationController

To delete a ReplicationController, the user needs to reference the latest namespace or ReplicationController transaction as input and a scriptSig value that satisfy the input transaction script. The value of the transaction must be null.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest ReplicationController
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": null,
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.7. Job

2.7.1. Create Job

To create a Job, the user needs to reference the latest namespace transaction as input and a scriptSig value that satisfies the input transaction script.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest namespace
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "Job",
              "metadata": {
                  "name": "my-job "
              },
              "template": {
                 “metadata”: {
                      "spec": {
                          "containers": {
                              “name”: "job-container",
                              “image”: "job-image",
                              “command”: [ “job”, “-parameter=1”
]
                          }
                      }
                  }
              }
          },
```

```
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.7.2. Suspend Job

To suspend a Job, the user needs to reference the latest namespace or Job transaction as input and a scriptSig value that satisfy the input transaction script. The value field must contain a fully updated Job document

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest Job transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "Job",
              "metadata": {
                  "name": "my-job "
              },
              "template": {
                 “metadata”: {
                      "spec": {
                          "containers": {
                              “name”: "job-container",
```

```
                                “image”: "job-image",
                                “command”: [ “job”, “-parameter=1”
]
                            }
                        }
                    }
                },
                "status": {
                    "conditions": {
                         "status": “True”,
                         "type": “Suspended”
                     }
                }
            },
            "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
        }
    ]
}
```

2.8. CronJob

2.8.1. Create CronJob

To create a CronJob, the user needs to reference the latest namespace transaction as input and a scriptSig value that satisfies the input transaction script.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest namespace
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "CronJob",
              "metadata": {
                  "name": "my-cronjob"
              },
              "spec": {
                 “jobTemplate”: {
                      “schedule”: “* * * * *”,
                      "spec": {
                          “template”: {
                              "containers": {
                                  “name”: "job-container",
                                  “image”: "job-image",
                                  “command”: [ “job”, “-
parameter=1” ]
                              }
                          }
                      }
```

```
                    }
                }
            },
            "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
        }
    ]
}
```

2.8.2. Delete CronJob

To delete a CronJob, the user needs to reference the latest namespace or CronJob transaction as input and a scriptSig value that satisfy the input transaction script. The value of the transaction must be null.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest CronJob transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": null,
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
```

```
  ]
}
```

2.9. ReplicaSet

2.9.1. Create ReplicaSet

To create a ReplicaSet, the user needs to reference the latest namespace transaction as input and a scriptSig value that satisfies the input transaction script.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest namespace
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "ReplicaSet",
              "metadata": {
                  "name": "nginx-deployment"
              },
              "spec": {
                  "replicas": 3,
                  "selector": {
                     “matchLabels”: {
                         “tier”: “front”
                     }
                  },
                  "template": {
                      “metadata”: {
                          “labels”: {
                              “name”: “development”
                          }
                      },
```

```
                        "spec": {
                            "containers": {
                                “name”: "nginx",
                                “image”: "nginx",
                                “ports”: {
                                    "containerPort": [ 80 ]
                                }
                            }
                        }
                    }
                }
            },
            "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
        }
    ]
}
```

2.9.2. Update ReplicaSet

To update a ReplicaSet, the user needs to reference the latest namespace or ReplicaSet transaction as input and a scriptSig value that satisfy the input transaction script. The value field must contain a fully updated ReplicaSet document.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest ReplicaSet
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
```

```
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "ReplicaSet",
              "metadata": {
                  "name": "nginx-deployment"
              },
              "spec": {
                  "replicas": 3,
                  "selector": {
                     “matchLabels”: {
                         “tier”: “front-release-candidate”
                     }
                  },
                  "template": {
                      “metadata”: {
                          “labels”: {
                              “name”: “release-candidate”
                          }
                      },
                      "spec": {
                          "containers": {
                             “name”: "nginx",
                             “image”: "nginx",
                             “ports”: {
                                 "containerPort": [ 8080 ]
                             }
                          }
                      }
                  }
              }
          },
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
```

```
OP_CHECKSIG",
      }
  ]
}
```

2.9.3. Remove ReplicaSet

To delete a ReplicaSet, the user needs to reference the latest namespace or ReplicaSet transaction as input and a scriptSig value that satisfy the input transaction script. The value of the transaction must be null.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest ReplicaSet
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": null,
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.10. StatefulSet

2.10.1. Create StatefulSet

To create a StatefulSet, the user needs to reference the latest namespace transaction as input and a scriptSig value that satisfies the input transaction script.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest namespace
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "StatefulSet",
              "metadata": {
                  "name": "storage-deployment"
              },
              "spec": {
                  "replicas": 3,
                  "selector": {
                     “matchLabels”: {
                         “tier”: “data”
                     }
                  },
                  "template": {
                      “metadata”: {
                          “labels”: {
                              “tier”: “storage”
                          }
                      },
```

```
                        "spec": {
                            "containers": {
                                “name”: "db",
                                “image”: "db",
                                “ports”: {
                                    "containerPort": [ 80 ]
                                },
                                “volumeMounts”: [ {
                                        "name": “db”,
                                        "mountPath":
“/usr/share/data”
                                   }
                               ]
                           }
                        },
                        "volumeClaimTemplates": [
                           {
                              "metadata": {
                                "name": "db"
                              },
                              "spec": {
                                "accessModes": [
                                  "ReadWrite"
                                ],
                                "storageClassName": "my-db-class",
                                "resources": {
                                  "requests": {
                                    "storage": "100Gi"
                                  }
                                }
                              }
                           }
                         ]
                    }
                }
            },
            "scriptPubKey": "
OP_DUP
```

```
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.10.2. Update StatefulSet

To update a StatefulSet, the user needs to reference the latest namespace or StatefulSet transaction as input and a scriptSig value that satisfy the input transaction script. The value field must contain a fully updated StatefulSet document.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest StatefulSet
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
              "kind": "StatefulSet",
              "metadata": {
                  "name": "storage-deployment"
              },
              "spec": {
                  "replicas": 3,
                  "selector": {
                     “matchLabels”: {
                         “tier”: “data”
                      }
```

```
                    },
                    "template": {
                        “metadata”: {
                            “labels”: {
                                “tier”: “storage”
                            }
                        },
                        "spec": {
                            "containers": {
                                “name”: "db",
                                “image”: "db",
                                “ports”: {
                                    "containerPort": [ 8080 ]
                                },
                                “volumeMounts”: [ {
                                        "name": “db”,
                                        "mountPath":
“/usr/share/data”
                                    }
                                ]
                            }
                        },
                        "volumeClaimTemplates": [
                            {
                                "metadata": {
                                  "name": "db"
                                },
                                "spec": {
                                  "accessModes": [
                                    "ReadWrite"
                                  ],
                                  "storageClassName": "my-db-class",
                                  "resources": {
                                    "requests": {
                                      "storage": "150Gi"
                                    }
                                  }
                                }
```

```
                            }
                        ]
                    }
                }
            },
            "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
        }
    ]
}
```

2.10.3. Remove StatefulSet

To delete a StatefulSet, the user needs to reference the latest namespace or StatefulSet transaction as input and a scriptSig value that satisfy the input transaction script. The value of the transaction must be null.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest StatefulSet
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": null,
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
```

```
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.11. DaemonSet

2.11.1. Create DaemonSet

To create a DaemonSet, the user needs to reference the latest namespace transaction as input and a scriptSig value that satisfies the input transaction script.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest namespace
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              {
                "apiVersion": "apps/v1",
                "kind": "DaemonSet",
                "metadata": {
                   "name": "elasticsearch",
                   "namespace": "kube-system",
                   "labels": {
                     "k8s-app": "logging"
                  }
                },
                "spec": {
                  "selector": {
                    "matchLabels": {
                      "name": " elasticsearch"
                    }
                  },
                  "template": {
                    "metadata": {
```

```
                    "labels": {
                      "name": " elasticsearch"
                    }
                  },
                  "spec": {
                    "containers": [
                      {
                        "name": "elasticsearch",
                        "image": "fluentd:v2.5.2",
                        "resources": {
                          "limits": {
                            "memory": "200Mi"
                          },
                          "requests": {
                            "cpu": "100m",
                            "memory": "200Mi"
                          }
                        },
                        "volumeMounts": [
                          {
                            "name": "varlog",
                            "mountPath": "/var/log"
                          },
                          {
                            "name": "varlibcontainers",
                            "mountPath":
"/var/lib/containers",
                            "readOnly": true
                          }
                        ]
                      }
                    ],
                    "terminationGracePeriodSeconds": 30,
                    "volumes": [
                      {
                        "name": "varlog",
                        "hostPath": {
                          "path": "/var/log"
```

```
                          }
                        },
                        {
                          "name": "varlibdockercontainers",
                          "hostPath": {
                            "path": "/var/lib/docker/containers"
                          }
                        }
                      ]
                    }
                  }
                }
              },
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.11.2. Update DaemonSet

To update a DaemonSet, the user needs to reference the latest namespace or DaemonSettransaction as input and a scriptSig value that satisfy the input transaction script. The value field must contain a fully updated DaemonSetdocument.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest DaemonSet
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
```

```
    }
],
"vout": [
    {
        "value": {
            {
              "apiVersion": "apps/v1",
              "kind": "DaemonSet",
              "metadata": {
                 "name": "elasticsearch",
                 "namespace": "kube-system",
                 "labels": {
                   "k8s-app": "logging"
                }
              },
              "spec": {
                "selector": {
                  "matchLabels": {
                    "name": " elasticsearch"
                  }
                },
                "template": {
                  "metadata": {
                    "labels": {
                      "name": " elasticsearch"
                    }
                  },
                  "spec": {
                    "containers": [
                      {
                        "name": "elasticsearch",
                        "image": "fluentd:v2.5.3",
                        "resources": {
                          "limits": {
                            "memory": "250Mi"
                          },
                          "requests": {
                            "cpu": "100m",
```

```
                              "memory": "400Mi"
                            }
                          },
                          "volumeMounts": [
                            {
                              "name": "varlog",
                              "mountPath": "/var/log"
                            },
                            {
                              "name": "varlibcontainers",
                              "mountPath":
"/var/lib/containers",
                              "readOnly": true
                            }
                          ]
                        }
                      ],
                      "terminationGracePeriodSeconds": 30,
                      "volumes": [
                        {
                          "name": "varlog",
                          "hostPath": {
                            "path": "/var/log"
                          }
                        },
                        {
                          "name": "varlibdockercontainers",
                          "hostPath": {
                            "path": "/var/lib/docker/containers"
                          }
                        }
                      ]
                    }
                  }
                }
              }
            },
            "scriptPubKey": "
```

```
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.11.3. Remove DaemonSet

To delete a DaemonSet, the user needs to reference the latest namespace or DaemonSet transaction as input and a scriptSig value that satisfy the input transaction script. The value of the transaction must be null.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest DaemonSet
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": null,
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.12. Secret

2.12.1. Create Secret

To create a Secret, the user needs to reference the latest namespace transaction as input and a scriptSig value that satisfies the input transaction script.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest namespace
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
                "data": {
                  "username": "dGhlc2lz",
                  "password": "ZXhjZWxsZW50"
                },
                "kind": "Secret",
                "metadata": {
                  "name": "mysecret",
                  "namespace": "default",
                  "resourceVersion": "1",
                  "uid": "7c41dad2-fc54-11ec-b939-0242ac120002",
                  "labels": {
                        "name": " elasticsearch"
                  }
                },
                "type": "Opaque"
              }
          },
```

```
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.12.2. Update Secret

To update a Secret, the user needs to reference the latest namespace or Secret as input and a scriptSig value that satisfy the input transaction script. The value field must contain a fully updated Secret.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest Secret transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
                "data": {
                  "username": "dGhlc2lz",
                  "password": "YXdlc29tZQ=="
                },
                "kind": "Secret",
                "metadata": {
                  "name": "mysecret",
                  "namespace": "default",
                  "resourceVersion": "1",
```

```
                  "uid": "7c41dad2-fc54-11ec-b939-0242ac120002",
                  "labels": {
                        "name": " elasticsearch"
                  }
                },
                "type": "Opaque"
              }
          },
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.12.3. Remove Secret

To delete a Secret, the user needs to reference the latest namespace or Secret transaction as input and a scriptSig value that satisfy the input transaction script. The value of the transaction must be null.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest Secret transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": null,
          "scriptPubKey": "
OP_DUP
```

```
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.13. ServiceAccounts

2.13.1. Create ServiceAccount

To create a ServiceAccount, the user needs to reference the latest namespace transaction as input and a scriptSig value that satisfies the input transaction script.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest namespace
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
              "apiVersion": "v1",
               "kind": "ServiceAccount",
               "metadata": {
                  "name": "my-service-account",
               }
          },
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.13.2. Remove ServiceAccount

To delete a ServiceAccount, the user needs to reference the latest namespace or ServiceAccount transaction as input and a scriptSig value that satisfy the input transaction script. The value of the transaction must be null.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest ServiceAccount
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": null,
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.14. Ingress

2.14.1. Create Ingress

To create an Ingress, the user needs to reference the latest namespace transaction as input and a scriptSig value that satisfies the input transaction script.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest namespace
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
            "apiVersion": "v1",
            "kind": "Ingress",
            "metadata": {
              "name": "minimal-ingress",
              "annotations": {
                "nginx.ingress.kubernetes.io/rewrite-target":
"/"
              }
            },
            "spec": {
              "ingressClassName": "nginx-example",
              "rules": [
                {
                  "http": {
                    "paths": [
                      {
                        "path": "/healthcheck-path",
                        "pathType": "Prefix",
```

```
                        "backend": {
                          "service": {
                            "name": "test",
                            "port": {
                              "number": 80
                            }
                          }
                        }
                      }
                    ]
                  }
                }
              ]
            }
          },
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.14.2. Update Ingress

To update an Ingress, the user needs to reference the latest namespace or Ingress as input and a scriptSig value that satisfy the input transaction script. The value field must contain a fully updated Ingress.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest Ingress transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
```

```
      }
  ],
  "vout": [
      {
          "value": {
            "apiVersion": "v1",
            "kind": "Ingress",
            "metadata": {
              "name": "minimal-ingress",
              "annotations": {
                "nginx.ingress.kubernetes.io/rewrite-target":
"/"
              }
            },
            "spec": {
              "ingressClassName": "nginx-example",
              "rules": [
                {
                  "http": {
                    "paths": [
                      {
                        "path": "/healthcheck-path",
                        "pathType": "Prefix",
                        "backend": {
                          "service": {
                            "name": "test",
                            "port": {
                              "number": 8080
                            }
                          }
                        }
                      }
                    ]
                  }
                }
              ]
            }
          },
```

```
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.14.3. Remove Ingress

To delete an Ingress, the user needs to reference the latest namespace or Ingress transaction as input and a scriptSig value that satisfy the input transaction script. The value of the transaction must be null.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest Ingress transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": null,
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.15. NetworkPolicy

2.15.1. Create NetworkPolicy

To create a NetworkPolicy, the user needs to reference the latest namespace transaction as input and a scriptSig value that satisfies the input transaction script.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest namespace
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": {
            "kind": "NetworkPolicy",
            "metadata": {
              "name": "test-network-policy",
              "namespace": "default"
            },
            "spec": {
              "podSelector": {
                "matchLabels": {
                  "role": "db"
                }
              },
              "policyTypes": [
                "Ingress",
                "Egress"
              ],
              "ingress": [
                {
                  "from": [
```

```
                {
                  "ipBlock": {
                    "cidr": "172.17.0.0/16",
                    "except": [
                      "172.17.1.0/24"
                    ]
                  }
                },
                {
                  "namespaceSelector": {
                    "matchLabels": {
                      "project": "myproject"
                    }
                  }
                },
                {
                  "podSelector": {
                    "matchLabels": {
                      "role": "frontend"
                    }
                  }
                }
              ],
              "ports": [
                {
                  "protocol": "TCP",
                  "port": 6379
                }
              ]
            }
          ],
          "egress": [
            {
              "to": [
                {
                  "ipBlock": {
                    "cidr": "10.0.0.0/24"
                  }
```

```
                    }
                  ],
                  "ports": [
                    {
                      "protocol": "TCP",
                      "port": 5978
                    }
                  ]
                }
              ]
            }
          }.
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```

2.15.2. Update NetworkPolicy

To update a NetworkPolicy, the user needs to reference the latest namespace or NetworkPolicy as input and a scriptSig value that satisfy the input transaction script. The value field must contain a fully updated NetworkPolicy.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest NetworkPolicy
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
```

```
"vout": [
    {
        "value": {
          "kind": "NetworkPolicy",
          "metadata": {
            "name": "test-network-policy",
            "namespace": "default"
          },
          "spec": {
            "podSelector": {
              "matchLabels": {
                "role": "db"
              }
            },
            "policyTypes": [
              "Ingress",
              "Egress"
            ],
            "ingress": [
              {
                "from": [
                  {
                    "ipBlock": {
                      "cidr": "172.17.0.0/16",
                      "except": [
                        "172.17.1.0/24"
                      ]
                    }
                  },
                  {
                    "namespaceSelector": {
                      "matchLabels": {
                        "project": "myproject"
                      }
                    }
                  },
                  {
                    "podSelector": {
```

```
                        "matchLabels": {
                          "role": "frontend-release-candidate"
                        }
                      }
                    }
                  ],
                  "ports": [
                    {
                      "protocol": "TCP",
                      "port": 6379
                    }
                  ]
                }
              ],
              "egress": [
                {
                  "to": [
                    {
                      "ipBlock": {
                        "cidr": "10.0.0.0/24"
                      }
                    }
                  ],
                  "ports": [
                    {
                      "protocol": "TCP",
                      "port": 5978
                    }
                  ]
                }
              ]
            }
          }.
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
```

```
OP_CHECKSIG",
      }
  ]
}
```

2.15.3. Remove NetworkPolicy

To delete a NetworkPolicy, the user needs to reference the latest namespace or NetworkPolicy transaction as input and a scriptSig value that satisfy the input transaction script. The value of the transaction must be null.

```
{
  "version": 1,
  "vin": [
      {
          "txid": "<txid from the latest NetworkPolicy
transaction>",
          "vout": 0,
          "scriptSig": "<ECDSA Tx Signature><PublicKey>"
      }
  ],
  "vout": [
      {
          "value": null,
          "scriptPubKey": "
OP_DUP
OP_HASH160
<hash_public_key_from_admin>
OP_EQUALVERIFY
OP_CHECKSIG",
      }
  ]
}
```