NOIA Network - Technical paper

Decentralized Content Delivery Network

Version 0.1

TABLE OF CONTENTS

Version 0.1 1

TABLE OF CONTENTS 2

INTRODUCTION 4

CONTENT SCALING LAYER 6

- CSL Organisation 6
  - CSL Cloud Controller Model 6
- Cloud Controller 7
  - Authorisation Control 7
    - Master Node Qualification process 7
  - Scaling Control 7
    - Load Balancing 7
      - Deterministic Content Scaling 8
      - Optimized Content Scaling 9
  - Data Collection and Performance Metrics 10
- CSL Master Nodes 11
  - CSL Master Node Client 11
    - Execution of Content Scaling Instructions 12
    - Issuance of Content Caching instructions 13
    - Fast Route Discovery 13
    - Merging of Requests 16
- NOIA Worker Nodes 17
  - Supported Storage Backends 17
  - CSL Worker Node Client 18
- CSL JS Library 19
      - Content Tagging 19

CSL WORKFLOW 21

GOVERNANCE RULES 23

- NOIA Monetary Rules 23
  - Choice of Token 23
  - NOIA Wallet Account 23
- NOIA Marketplace 24
  - Hiring Process 24
  - Work & Reward Process 25
  - Smart Contracts Details 26
    - NOIA User Contract 27
    - NOIA Worker Node Contract 27

NOIA Work Contract 27
NOIA Work Center 28
NOIA Courtroom 28
CSL Marketplace 28
CSL Master Node Qualification Process 28
CSL Master Node Work Distribution Process 29
Smart Contracts Details 29
CSL Cloud Controller Contract 29
CSL Master Node Contract 29
CSL Subcontract 30

NOIA Token economy 31
Worker node reward system 31
NOIA Payment gateway 32

References 33

INTRODUCTION

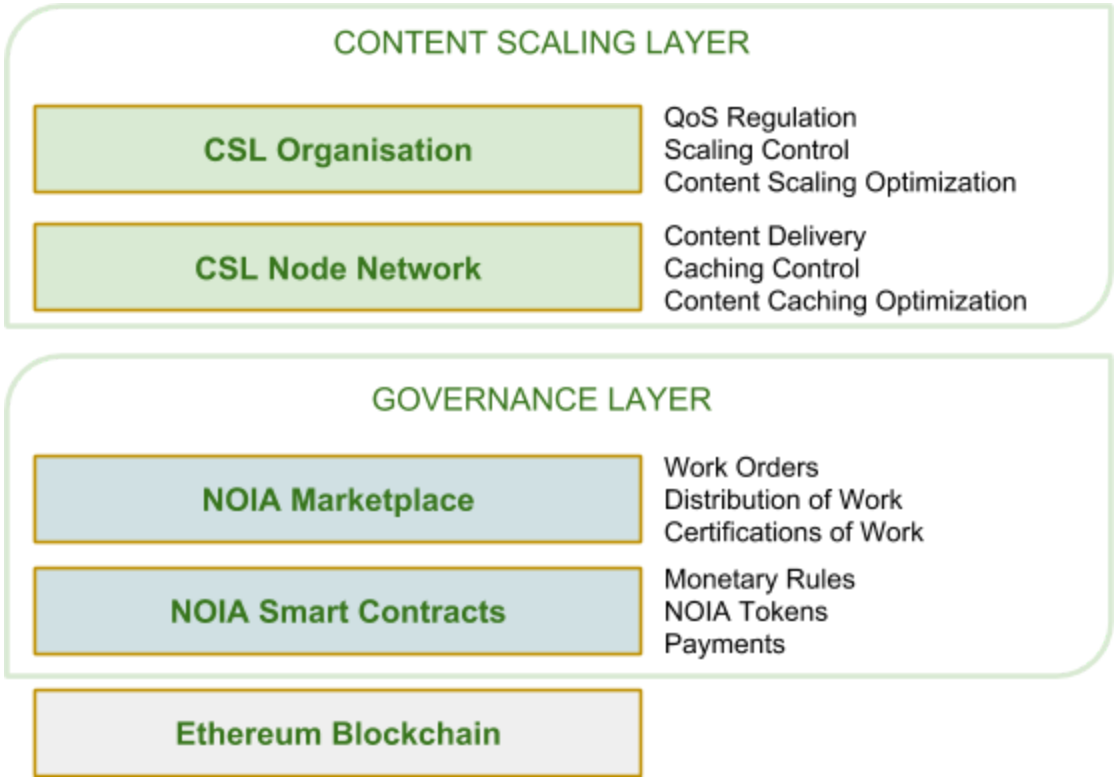
NOIA Network is the next-generation decentralized Content Delivery Network, utilizing unused storage and bandwidth from computers around the world, to create a more widely distributed, efficient and integral layer of internet infrastructure for global content delivery.

NOIA's architecture is formed by combining two separate technological elements:

- **Content scaling layer (CSL):** A combination of peer-to-peer (P2P) file scaling protocols, artificial intelligence (AI) and blockchain technologies that determine how content is delivered through the network.
- **Governance layer:** A set of smart contracts that define and govern NOIA tokens and how NOIA tokens are transferred between NOIA worker nodes and their users in the content delivery economy.

Quality of service and optimal content scaling is governed by CSL Organisation, which is at the top-most level in the technology stack and its implementation can be centralised as well as decentralised.

Figure 1: NOIA Technology stack



Popular content is cached on CSL node network, comprised of master nodes and worker nodes. Node network aims to satisfy as many user requests for the content as possible and minimize the amount of content delivered from the storage backend – the original source of the content.

Both CSL organisation and CSL node network are assisted by machine intelligence algorithms to make their performance optimal.

NOIA marketplace ensures that master and worker nodes are rewarded for their services to the society: scaling, caching, and delivering the content through the web. This is achieved with a help of smart contracts implemented on top of the Ethereum blockchain.

The unique combination of content scaling and governance layers allows for optimal use of node network resources simultaneously rewarding these nodes for their performance.

NOIA creates a win-win situation for all the parties involved: users, websites, and nodes that make up the network. By purchasing the untapped technological potential of the swarm, NOIA promises to optimize the functioning of the internet – websites pay less for their content delivery service while users enjoy a faster and more efficient web.

CONTENT SCALING LAYER

CSL Organisation

A system that subcontracts a number of NOIA master nodes to help the CSL organise NOIA worker nodes efficiently to scale content delivery to its customers.

Features of CSL Organisation can be grouped into 3 major groups:

- Authorisation control and quality of service regulation
- Content scaling control (including load balancing and master node selection)
- Data collection and data-driven optimisation

There are two models of how Content Scaling Layer Organisations can be implemented on the NOIA Network:

- CSL Cloud Controller Model – a single centralised point in NOIA Network which controls other CSL system layers
- CSL DAO Concept Model – a fully decentralised organisation with self-imposed governance

Launching NOIA as a fully decentralised platform from day one poses a risk of the technology being not mature or ready enough to gain traction and wide adoption. There are also technological limitations on available decentralised infrastructures or they aren't mature enough to implement the CSL organisation on a decentralized cloud, even if NOIA CSL would be ready in terms of performance and quality. Therefore, NOIA will launch using a centralised Cloud Controller to ensure system availability and quality as well as having the required control to improve the technology and CSL Layer. Once a scalable decentralised cloud becomes a reality, NOIA Network can be transformed into a fully Decentralised Autonomous Organisation. Further discussion in this paper will assume the use of CSL CLoud Controller as an Organisation Model, unless specified otherwise.

CSL Cloud Controller Model

The CSL cloud controller is a cloud service implementation of the CSL organisation. The cloud service can be run on any of the public cloud platforms, including AWS, Azure, Google Compute Engine or the NOIA privately owned infrastructure.

The main control and data flows of the CSL cloud controller are:

- The controller listens to browser requests and helps browser select the master nodes.
- The controller listens to master nodes for their status updates.

- Based on status updates from master nodes and browser requests, the controller independently sends scaling instructions to the master nodes, with the help of machine intelligence algorithms.
- The controller detects and de-lists faulty or fraudulent master nodes, to assure the level quality of service promised to the websites.

Cloud Controller

Authorisation Control

Here are more detailed descriptions of how the *CSL cloud controller* implements the key features of a *CSL organisation*.

Master Node Qualification process

The *Cloud controller* takes care of the authorization of *master nodes* as well. Master nodes have to comply with the requirements as set forth in the Governance Layer. Moreover, they have to qualify and satisfy both physical and economic requirements, such as strategically attractive geolocations and willingness to guarantee scaling the amount of content set forth in the CSL subcontract (see CSL Subcontract). The qualification process may contain a set of system tests, a deposit as proof of stake, etc.

Scaling Control

The *Cloud controller* is the key element in the centralised version of the CSL organisation model. The main function of the Cloud controller is to give *content scaling instructions* to the master nodes, which means that there is a demand for a certain amount of content to be scaled and it has to be fulfilled by the master nodes giving *content caching instructions* to worker nodes.

To give content scaling instructions, the cloud controller requires quantifiable data, i.e. metrics about the past performance of the master nodes and worker nodes. Machine intelligence (MI) algorithms trained on these data results in optimized functioning of the CSL. Key areas where MI is applied include load balancing, smart caching, traffic prediction, and traffic anticipation.

Load Balancing

Whenever a *user* enters a website with CSL support, the *cloud controller* is also responsible for selecting and assigning the *master node* closest to the *user*. Initially, the *Master node* can be selected only by its geolocation and the current workload factor – idle *master nodes* close to the *user* are generally preferable over busy and remote *master nodes*.

Over time, master node selection will become optimized and continuously re-optimized with the help of MI algorithms. Historical patterns showing demand dynamics and swarm connectivity in specific locations will become a basis for informed load balancing decisions.

Historical data about the master nodes and their performance are used to: (1) estimate their health and ability to scale specific content at specific times; (2) select a master node to serve an incoming request.

There are two main goals we are trying to achieve here: (1) keep the system balanced, i.e. on average all nodes should expect about the same service load; (2) deliver content as fast as possible. At the same time, we are trying to ensure that all scaling contracts are actively used, so that no master node is forgotten, which could happen without such optimization (e.g. due to its remote location).

Deterministic Content Scaling

The Content Scaling Layer does not function as a typical CDN: it only stores content for a limited and – and even more importantly – for an optimal time. Smart caching strategy is essential for proper scaling functionality. The following presents a prototype of a such strategy, not based on pattern recognition algorithms, but is instead rather a predefined set of rules. Thus, it is very fast and does not require lengthy processing of historical data.

The file's availability on the swarm is a function of its short-term popularity. The Cloud controller / master node is tracking the number of requests per file. As requests are time-stamped, it is easy to construct a sliding window statistic of r_Δ – *number of requests per* Δ *time*. Here Δ is the length of a sliding window, dependent on the type of content. It is constantly being updated by:

$$r_\Delta := r_\Delta + r_{new} - r_{old},$$

where r_{new} are the number of new incoming requests, and r_{old} are the number of requests older than Δ.

To make this caching strategy efficient and be robust against sporadic requests, there must be some minimum number of requests r_{min} originating from different users. Only after this minimum is reached within Δ time, is the file is initially downloaded to the CSL. Let's denote r_{last} as the number of requests when the *initial caching* decision or every subsequent *scaling-up* decision, i.e. decision to increase the availability of the file in CSL, is made. Then, every *scaling-up* is performed when the following is true:

$$r_\Delta \geq C_{scale-up} \times r_{last},$$

where $C_{scale-up} > 1$ is a parameter regulating the speed of spreading the content in the CSL. Suppose $r_{min} = 8$ and $C_{share} = 2$, which results in N_{init} nodes caching the file and $r_{last} = 8$. If within the next Δ seconds r_{Δ} grows to 16, we share the file with additional N_{add} nodes. If it further grows to 32, we share with N_{add} more additional nodes, etc.

Similarly, we implement the removing from cache or *scaling-down* decisions. A file is removed from a certain number of the nodes if:

$$r_{\Delta} \leq C_{scale-down} \times r_{last},$$

where $C_{scale-down} < 1$ is a parameter regulating the speed of content uncaching. When r_{Δ} becomes zero, a file is removed from all the nodes still storing it at that point.

Optimized Content Scaling

When the content is hosted through the CSL for a period of time, advanced machine intelligence algorithms can be applied to gain actionable insights from the historical data traffic and assist in scaling instructions given by the *cloud controller* to the *master nodes*.

Collecting the file metadata and associated requests will help the *cloud controller* to explain why some files are popular, and some are not. Files with similar metadata to the historically popular files can be classified as potentially popular as well, hence their presence in the swarm can be increased upfront.

Under the assumption that the demand for viral (i.e. extremely popular) files have a trend and seasonality, we can manage the delivery of those files effectively, thus satisfying the demand in an optimal and cost-effective way.

Alternatively, analysing the historical throughput data would help us optimize our routing methodology. It boils down to to answering the question “which files are popular where and at what time?”, by moving those files to the closest nodes (cf. *points of presence* in a CDN network) at the right time. This sort of load balancing is what distinguishes CSL from a standard CDN network.

Holding popular files in the cache incentivizes the CSL worker nodes to participate in the system, as it increases their chances to earn tokens faster. A caching strategy based on historical pattern recognition would be directed to ensure the optimal functioning of the CSL layer for the best interests of customers as well as worker nodes.

Examples:

- An audio podcast is released every Thursday 9 PM, it attracts thousands of listeners with popularity peaking Friday morning.
- There's an ongoing political scandal in US. So far, most of the videos covering it attracted tens of thousands viewers in the first hour after release.

Data Collection and Performance Metrics

The collection of time series data is necessary for optimized functioning of the CSL and its Scaling Control. Performance metrics reported by *worker nodes* and derived by *master nodes* mainly act as constraints on these optimization tasks.

List of metrics reported by *CSL worker node client* upon joining the swarm:

- CPU model (e.g. INTEL® CORE™ i7-8700K)
- RAM specifications (e.g. DDR4-2133)
- Hard Disk Drive type, which contains partition used for CSL (e.g. SSD, 'D:/')
- Network adapter specifications (e.g. Killer™ Ethernet E2500)

List of raw metrics periodically sent by *CSL worker node client* to *CSL master node client* under a certain smart contract:

- Data hash signatures
- Storage dedicated to CSL
- Storage used by CSL
- Bandwidth dedicated to CSL
- Bandwidth used by CSL
- Node uptime

These metrics become the basis for node employment while entering into a smart contract. They act as constraints of what can be stored at and routed through the worker node.

The following is a list of derivative metrics periodically calculated by the *CSL master node client* and sent together with the raw metrics to the *cloud controller*:

- Node storage utilization
- Node bandwidth utilization
- Node employment rating – a function dependent on storage and bandwidth utilization
- Node health – a function dependent on storage and bandwidth utilization
- Request dynamics (as described in Deterministic scaling)
- File ownership, i.e. a list of hashes
- File popularity score

These data are collected and constantly mined by the *cloud controller*. Thus, any new master node joining the CSL can be initially instructed based on the historical performance of other master nodes.

CSL Master Nodes

They are independent services run by entities that are capable of operating high SLA services.

They are subcontracted by the *CSL Cloud Controller or CSL DAO - QoS Control rules*, once it's passed the qualification process.

They should be compatible with the protocol that is supported by the *CSL Controller*. Therefore NOIA will release two Open Source projects:

- *CSL master node reference client* – a reference client that one could use to host their own master node.
- *CSL Master Node SDK* – a helper library that helps one develop one's own CSL master node client.

CSL Master nodes can freely decide their own pricing model in order to compete with other master nodes with a good combination of QoS level and price.

Main features of a master node include:

- Load balancing: monitor worker node load, and distribute load amongst them by issuing content caching instructions
- Fast route discovery: select worker nodes that are fastest for the browser to download from

NOIA will launch using the CSL Cloud Controller Model and a set of our hosted CSL Master Nodes which pass quality requirements and ensure network stability, before Open Source SDK projects and the whole network is ready to accept independently-run Master Nodes.

CSL Master Node Client

The *CSL master node client* is a proprietary application, developed and owned by NOIA. It is responsible for the transport of requests, submitting performance metrics to the cloud controller, fastest route discovery, request pooling and smart cache management. The master node has access to the database with all the logs of cached content in the swarm.

The *CSL master node client* is built on top of UDP Tracker Protocol for BitTorrent. Functioning as a sort of BitTorrent tracker, it submits the *shortlist* of node addresses for file download either to a *user* (fulfilling a request) or to a *worker node* (sharing a content in the swarm).

Metrics derived from the file requests and information sent by the *CSL worker node clients*, are sent to the *cloud controller*, which performs most of the processing power intensive optimization pertaining to the scaling of the content. However, a *master node client* makes the following decisions with the help of statistical modelling and machine learning algorithms:

- Initial caching of the file in CSL: when? which nodes?
- Additional caching of the files in CSL: which nodes? which files? when?
- Deleting the files from the nodes' cache: which nodes? which files? when?

A *master node client* initializes these processes, whereas, *worker node clients* fulfil them.

The *shortlist* of worker nodes for content delivery is optimized by a master node. Finding this optimal list can be viewed as a *weighted minimum coverage set* problem. The list always includes at least k worker nodes for the same segment, hereby minimizing the risk of slow or failed delivery, if some worker nodes in the list suddenly disconnect from the network. The nodes are weighted with respect to their proximity to the requesting *user*, thus ensuring the fastest possible content delivery.

The *CSL master node client* keeps a recent history of requests and also send these statistics to the *cloud controller*. Requests are pooled saving response time and processing power (load balancing). Response to a number requests over a short period of time may be identical. In case there are many requests for the same data coming from multiple users, who share a similar geolocation, such pooling is trivial.

To manage the cache available on worker nodes, the CSL master node client keeps track of request dynamics. As the number of requests associated with one file is increasing, selected worker nodes in the swarm are instructed to cache the file by downloading it from the swarm. Decline in the number of requests causes the file being deleted from the cache of selected worker nodes.

Execution of Content Scaling Instructions

The master node client does not only fulfill requests, it also accepts and executes direct content scaling instructions from the cloud controller. These instructions are based on machine intelligence algorithms and are meant to optimize the CSL with respect to load balancing and smart caching.

The master node client receives a *content scaling instruction* from the cloud controller in the form of a list of content hashes. The master node client starts executing the scaling-up instruction by selecting worker nodes for the task and issuing *content caching instructions* to these nodes.

Note, content delivery to users has priority over fulfilling caching requests. The Cloud controller sends content scaling-up instructions with some time reserve in mind, as caching bigger files might take some time.

Removing content from the cache, or scaling-down instructions, follows the same logic. The master node client receives one instruction from the cloud controller and subsequently distributes the tasks to worker nodes.

Issuance of Content Caching instructions

An additional challenge to content scaling instructions given by the cloud controller is deciding, which worker nodes should cache the file or remove it from their caches. This is done by the master node. Several variables that play a key role here are (1) the location metadata of incoming requests, (2) the location metadata of nodes, (3) health of the nodes as a function of their resource utilization.

We seek to improve the availability of files within geo-locations, where most of the requests are coming from. However, some of them will be utilizing less of their storage or bandwidth dedicated to the CSL. Thus, during the *scaling-up*, we are targeting the healthiest nodes (having largest unutilized bandwidth and largest amount of unutilized storage), whereas, during the *scaling-down*, we are targeting the least healthy nodes (having the smallest unutilized bandwidth and smallest amount of unutilized storage).

Worker nodes cache the content either from their neighbours in the swarm, or from original backends. Once a file is initially cached in the swarm, nodes can share the file between themselves. A master node instructs selected nodes by sending the list of nodes holding the file in their storage and file metadata. Alternatively, worker nodes can employ DHT protocol to discover neighbours holding the content in a decentralized way.

Fast Route Discovery

Master node clients ensure optimal routing in the CSL. The route speed and resulting throughput can be defined as a function of *latency* between a user requesting the content and a node fulfilling this request, as well as the *uplink bandwidth* of the node. Note, that we cannot account for the downlink bandwidth of a user.

The selection of the nodes in the swarm with the lowest latency to a certain user is trivial, when there exists a sufficient set of such nodes. The size of such a set would be determined by the byte size of the content requested.

However, suppose the file is large and there is only one node holding this content in close proximity to the user. In such a case, it could be practical to cache the file in the storage of other nodes close to the customer. While the request is being fulfilled from a single node in the beginning, more nodes would join the peer list, as they are discovered by the customer using the peer exchange protocol.

The *K-Shortest Paths* algorithm helps answer the question, which nodes should be participating in rerouting of the content to the user. The K-Shortest Paths in this case can be viewed as a generalized version of *Dijkstra's single-source shortest path algorithm* (SSSP) for *min-priority queues*.

Dijkstra's algorithm employing a min-priority queue is the asymptotically fastest known algorithm for *directed graphs* with time complexity of $O(|C| + |N|log|N|)$. In our case $|N|$ is the number of worker nodes and $|C|$ is the number of feasible p2p connections between worker nodes and the customer. Here, *feasible p2p connections* means that, in the route graph, not every node is directly connected to all other nodes. Lowering $|C|$ helps us to reduce the complexity of the algorithm. Chen et al. (2007) find that the simpler version of this algorithm – heap without support to decrease of priority operation – results in the best performance.

Pseudo code of Dijkstra's SSSP algorithm without decrease of priority operation:

$G :=$ directed graph with vertices and edges connecting them

$Q :=$ vertices of the G stored in heap data structure, e.g. the most popular is binary heap

$V(G) :=$ vertex set of the graph

$s,\ u,\ v \in V(G)$ are vertices in the graph – vertex s is called *source vertex*, whereas, u and v are any other vertices. Vertices are stored in a heap as (u, k), where $k \in \mathbb{R}$ is accumulated priority (or distance)

$E(G) :=$ edge set of the graph, e.g. edge between u and v is denoted by (u, v)

$w : E(G) \rightarrow k \in \mathbb{R}$. Weight (priority / distance) function, e.g. weight associated with edge (u, v) is denoted by $w(u, v)$

$d[u] :=$ result vector for storing distances of each vertex u

$Insert$ and $ExtraxtMin$ are regular operations for any heap data structure.

Function, which computes the shortest distance from s to each vertex $v \in V(G)$:

$Q \leftarrow \varnothing$ // initialize empty heap

for each $v \in V(G)$ ***do*** $d[u] \leftarrow \infty$ // initialize all distances as unknown

$Insert(Q, (s, 0))$ // add source vertex to the heap

while $Q \neq \varnothing$ ***do***

$(u, k) \leftarrow ExtraxtMin(Q)$ // extract vertex with min priority

if $k < d[u]$ ***then*** // if distance is less than what is stored

$d[u] \leftarrow k$ // update accumulated distance

for each $(u, v) \in E(G)$ ***do*** // *v* are all neighbouring vertices to *u*

if $d[u] + w(u, v) < d[v]$ ***then*** // compare distances

$Insert(Q, (v, d[u] + w(u, v))$ // insert v to the heap

$d[u] \leftarrow d[v] + w(u, v)$ // update accumulated distance

In our case we will stop the above function when a certain number of nodes (vertices) holding the content required by customer (source vertex) is assigned in $d[u]$. The algorithm can be implemented with various types of efficient heap data structures. Adoption of this SSSP algorithm in the CSL framework and selection of the implementation method will be based on empirical test results, such as those in Chen et al. (2007).

As routes in the graph are directed towards the customer (source vertex s), nodes can be organized in layers depending on the mileage between them and the customer. Nodes in one layer can only connect between themselves and to the nodes in neighbouring layers. So, e.g. node in layer 3 can connect to nodes in layers 2, 3, and 4, but not to nodes in layers 1, 5 and 5+.

In a trivial case, the cost of routing is equal to mileage between user and worker nodes. However, the priority in the queue can be expressed as function on both mileage and available uplink bandwidth.

Implementing min-priority queues as heap data structures allows for the efficient discovery of an element with min-priority with complexity of $O(1)$, while insertion and deletion are also very fast with time complexity of only $O(log|N|)$.

Merging of Requests

CSL is by itself an internet layer designed for pooling of requests to significantly decrease the number of requests directed to the storage backends.

Given a request for a single file, the selection of closest worker nodes neighbouring a user is trivial. However, further merging of requests can performed based on their origin and similarity. Master node clients are responsible for merging requests that (1) originate from the same user during a very short window of several milliseconds, or (2) originate from different but similar in their geolocation users during a slightly longer time window.

Examples:

- (1) A customer enters a website offering to download a set of softwares required for performing some task. There is an one-click option to download them all. A real-time response is arranged by a master node to download all those files at once from the swarm. The response time lag is expected by a customer, since collecting all files usually takes time as compared to downloading a single file.
- (2) Every time a browser enters website exactly the same set of images is loaded. Until this set expires (e.g. a new image is added to the website), a number of optimal request responses can be predefined for multiple geolocations.

The merging of requests to derive an optimal list for response sent by a master node can be interpreted as a weighted minimum coverage set problem, where files are split into packets of the same size and nodes are usually holding overlapping sets of such packets. Additionally we want to choose those nodes that are closer to the customer, and ideally the list should contain at least k nodes per each packet. The problem is NP-Hard, however, it can be approximated by a greedy algorithm to achieve $1-\frac{1}{e}\approx 0.632$ approximation ratio, or by relaxing some constraints we could even achieve an approximation of $O(logN)$ factor (Abreu and Cardoso, 2014). The weighted minimum coverage set problem is formulated as a linear programming problem to find an optimal set $\widehat{S}$:

Minimize $\sum_{S_j \subset \Omega} w_j \cdot x_j$, subject to $\sum_{j:p_i \in S_j} x_j \geq k, \quad \forall p_i \in \Pi$, where:

$\Omega :=$ set of worker nodes holding at least one packet requested

$S_j :=$ set of packets held by worker node *j*

$\Pi :=$ superset of all packets in a merged request

$p_i :=$ packet *i*

w_j := weight associated with node *j* (a function of mileage between node *j* and customer)

$x_j \in \{0, 1\}$ is worker node selector: if $x_j = 1$, node *j* is selected into $\widehat{S}$

k := minimum number of each packet covered by $\widehat{S}$

While solving such a problem for a real-time response could be too slow due to its complexity, it can be stored in a memory for a short time as a ready-made response for exactly the same requests. As a result, request merging is performed in a sense that the CSL tries to anticipate upcoming requests.

NOIA Worker Nodes

These are an agent that can be hired by anyone through the NOIA Market to do content scaling related work. They run softwares that are not specific to CSL. They could be hired for other applications.

Worker nodes are independent network nodes that respond to caching instructions from the master node, and download requests from browsers. Their work is evaluated by the master nodes directly.

They are free to choose different master nodes to apply for jobs at. A NOIA worker node charges for the resources it shares. The hiring and reward process are described in the NOIA marketplace.

Supported Storage Backends

Customers can host their content in different backends that the CSL client worker nodes support, including:

- HTTP protocol, that means customers can either host their content on their own server which has an public IP, or on any CDN or any other network driver that expose original content in a http link.
- FTP protocol, a mature and aging but still efficient way of hosting contents supported by CSL worker nodes.
- IPFS protocol, for customers who prefer a decentralized network paradigm, IPFS is one of the popular choices. CSL can help them to scale the distribution of their files on IPFS, without worrying about content availability and throughput limit that using IPFS alone may face.

NOIA will release an Open source project **CSL Worker Node SDK**. Any CSL Worker Node Clients written in accordance with the SDK will be able to participate in the NOIA network and be compatible with its integral parts.

CSL Worker Node Client

The CSL worker node client is an open source Node.js application that can run on any Linux, Windows or MacOS machine. Using the CSL worker node clients people can rent out their unused bandwidth and storage space to participate in the CSL network to deliver web content from storage backends to users.

The CSL worker node client is responsible for downloading and caching the content from *storage backends*, sharing it to other *worker nodes* in the swarm, and most importantly, delivering it to the *users*.

The *CSL worker node client* caches (and also uncaches) the content only when instructed by a *master node client*. Such instruction is received in a form of a *shortlist* containing addresses of nodes in the swarm, who already store the content in their cache.

Downloaded content is saved in the *worker nodes*' cache – local storage dedicated for the use by the CSL. The *Worker node client* sends request to an original *storage backend* for a content using original link to the source, obtained from a *master node client*.

A peer-to-peer (p2p) protocol resembling BitTorrent is used for both (1) content delivery to the *users* and (2) content sharing between the *worker nodes*. Uploads to the user applications, e.g. browsers, always have priority over sharing among the nodes in the swarm, thus ensuring the efficient delivery of the content to the website customers.

Additionally, Peer Exchange (PEX) and Distributed Hash Table (DHT) protocols are used to enable a worker node to discover supplementary sources for the data it has to download from the swarm. They are especially useful, if nodes in the *shortlist*, received from a *master node*, disconnect from the swarm before the download is complete or are busy with other uploads and cannot deliver the best speed.

The *CSL worker node client* also sends a node's performance metrics to the *CSL master node client*, describing how the node handles a workload assigned to it through smart-contracts. Subsequently, these and other derivative metrics help the *cloud controller* to optimize the functioning of the whole CSL network.

CSL JS Library

The yet remaining piece of the puzzle is how a hosted website interacts with the NOIA network instead of going directly to the original source.

The CSL JS Library is an open source solution, which has to be integrated to the hosted website to use CSL. Its core functionality is to submit data requests to the master node and receive the data from the swarm. In case of swarm-related failure, it falls back to the storage backend – an original source of the content.

The links to the original source are changed by hash signatures of this content. Thus, a user application requesting some files is sending their hash signatures to the CSL master node, which in turn responds with an optimal peer list for downloading the content.

In case the file is not cached (e.g. it was never requested before or is missing from the swarm due to its scarcity), a master node will respond to request with an empty list of worker nodes. Subsequently, the CSL JS Library will fulfil the request by downloading the file directly from the storage backend.

Sporadic requests for a certain content should not result in the CSL caching this content. However, if the number of requests during a time window grows and reaches the preset minimum number, optimized per each piece of content by analysing historical requests, subsequent requests will be fulfilled by the swarm until the demand for that content subsides.

Content Tagging

There are two types of tags tracked by the CSL JS Library: (1) tags generated by website users and (2) tags created by website owners. User-generated tags assist in machine intelligence algorithms for optimal load balancing, while owner-generated tags result in deterministic content scaling instructions.

Examples of user-generated tags are the number of likes on Facebook, number of retweets in Twitter, or number of pins in Pinterest. User tags can be viewed as a type of content metadata.

The *Cloud controller* can forecast the file popularity based on the initial reaction to the content on social media. Contents of future events, popular in social media, are scheduled to be cached at the first instance of publishing in order to best fulfil the expected surge of requests. This type of scheduling based on insights from metadata can be referred to as *traffic anticipation*.

Examples of traffic anticipation:

- A webzine story is shared X times during the first Z minutes.

- Miscellaneous content with a certain hashtag is currently trending in social networks.

The CSL JS Library allows website owners to tag certain content in order to hire the *cloud controller* to cache resources for a certain time. This can be referred to as *scheduling on demand*. The CSL will be integrated with Google Analytics. Website owners can view what content was popular at what times and make their own *traffic anticipation* decisions.

When content is tagged by a website owner as important, the CSL JS Library passes scaling instructions for this content through the cloud controller to the swarm. However, this service requires dedicated storage capabilities, and it is likely to be sub-optimal from the general CSL functioning perspective, hence it will bear additional costs to website owners.

CSL WORKFLOW

A usual CDN network responds to a download request by allowing an end user to download online content from the closest Point of Presence (PoP) belonging to that CDN. This minimizes the latency between end user and storage backend. It also aims to deliver the best possible speed.

The Content Scaling Layer works in a very similar way by finding the closest and fastest nodes to fulfil a request. Originally all content still comes from some storage backend. However, the number of requests that backend has to fulfil is significantly reduced, as most of them are fulfilled by the CSL.

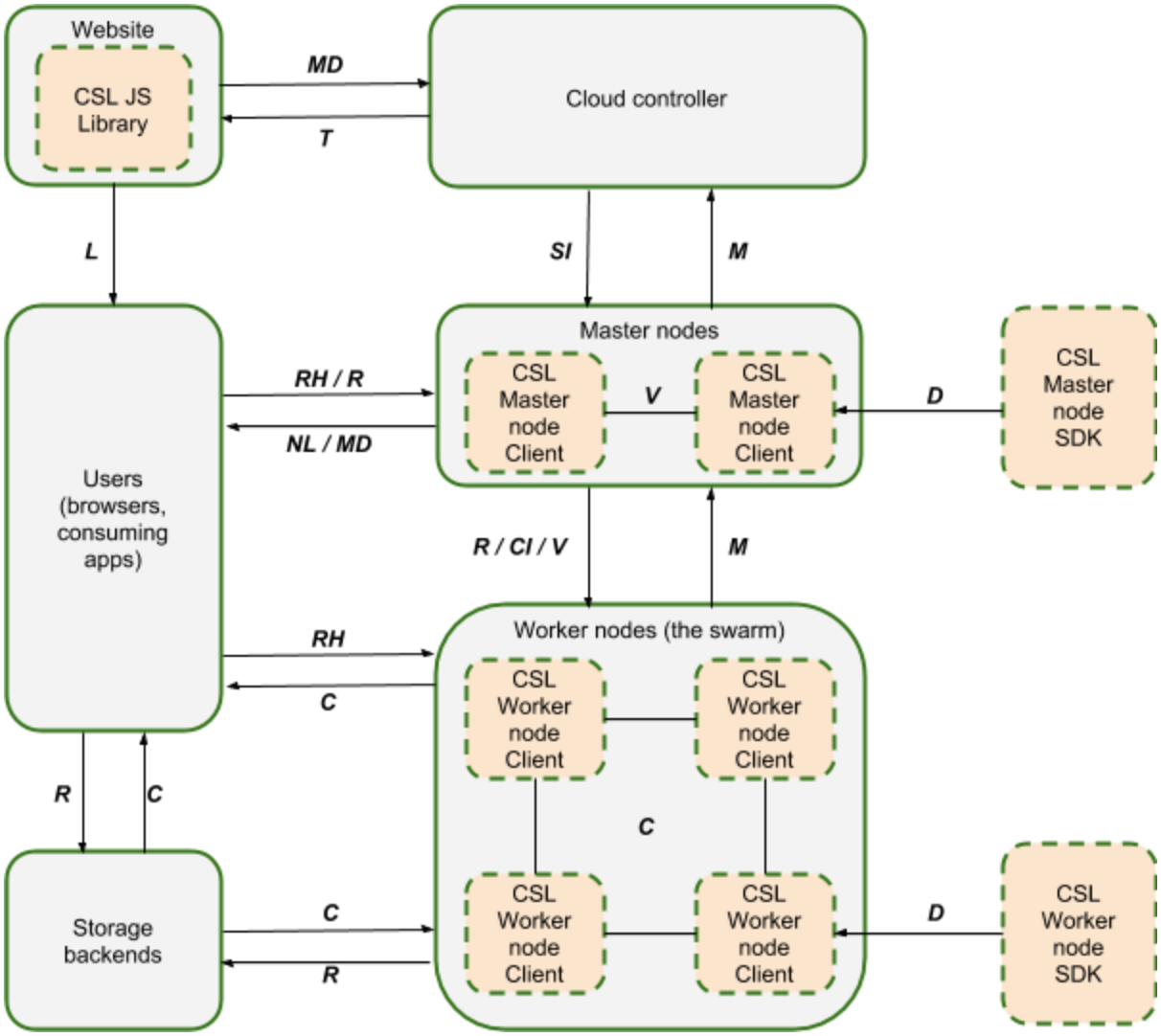
The general workflow of CSL:

1. A *user* enters the website which supports CSL. The *CSL JS Library* is loaded into a local memory of a consuming application (e.g. browser).
2. A *user* sends a request for a content. *CSL JS Library* translates request into SHA256 type of infohash.
3. The *CSL JS Library* passes the request and its infohash to the closest *master node client*, selected by the *cloud controller*.
4. A *master node client* checks if the requested content is in the swarm and responds to a *user*.
    A. If the content is *not* available, it responds with an empty list of nodes.
    B. If the content *is* available, it responds with the optimal list (*shortlist*) of nodes and metadata required for the content delivery.
5. A *user* starts downloading data from the nodes in the list using infohash. If an empty list is returned, it falls back to the original backend for content delivery.

Figure 2 shows the role and workflow between the aforementioned collections of CSL software.

Figure 2: CSL workflow diagram

R = sends request for content (source)
RH = sends hashed-request for content
C = sends or shares content
T = tracks incoming users
NL = sends Node List
MD = sends content metadata
CI = sends Caching Instructions
SI = send Scaling instructions
V = Validates
M = Reports performance metrics
D = Defines
L = loads JS Lib


Website
CSL JS Library
MD
T
Cloud controller
L
SI
M
Master nodes
CSL Master node Client
V
CSL Master node Client
D
CSL Master node SDK
Users (browsers, consuming apps)
RH / R
NL / MD
R / CI / V
M
Worker nodes (the swarm)
RH
C
CSL Worker node Client
CSL Worker node Client
C
R
C
CSL Worker node Client
CSL Worker node Client
Storage backends
C
R
D
CSL Worker node SDK

GOVERNANCE RULES

The governance rules in NOIA define how to authorise the usages of NOIA worker nodes and CSL master nodes:

- The *NOIA monetary rules* define what is a *NOIA token* and the wallet contract that can hold and transact the tokens.
- A *NOIA worker node* expects to be rewarded with *NOIA tokens* for sharing their resources to authorize its usage. The *NOIA Marketplace* defines the rules of how the *NOIA worker nodes* can be hired, rewarded and how disputes can be resolved.
- A *CSL master node* needs to be paid first in order to pay the *NOIA worker nodes* it manages. The *CSL Marketplace* defines the rules of how the *CSL Cloud controller* selects the *CSL master nodes*, and how the *NOIA tokens* are distributed to the *CSL master nodes*.

The governance rules are running on the Ethereum blockchain and written in Solidity smart contract language.

NOIA Monetary Rules

Choice of Token

NOIA needs a financial accounting unit for its contracts. In principal, the token system is independent of the NOIA. The choice of the token should be based on these principals:

- It is ERC20 compliant[1].
- It has a stable market value as a medium of exchange.

Considering the absence of such a token with a stable market value, Noia will need to issue its own tokens. Because the token will be backed by its utility value, that is, to buy services from the CSL, then the token will not run out of buyers as long as the CSL serves customers. The token will hence have a stable market value, when customers value the services provided by the CLS in most likely other major currencies.

NOIA Wallet Account

Since NOIA tokens are ERC20 compatible, an Ethereum wallet address is sufficient to act as a NOIA account suitable to launch a NOIA worker node. NOIA worker nodes will have an implementation of a NOIA wallet, which will be capable of receiving NOIA Tokens (sent between accounts or as rewards for participating in the NOIA network), creating your unique address

[1]https://github.com/ethereum/EIPs/issues/223

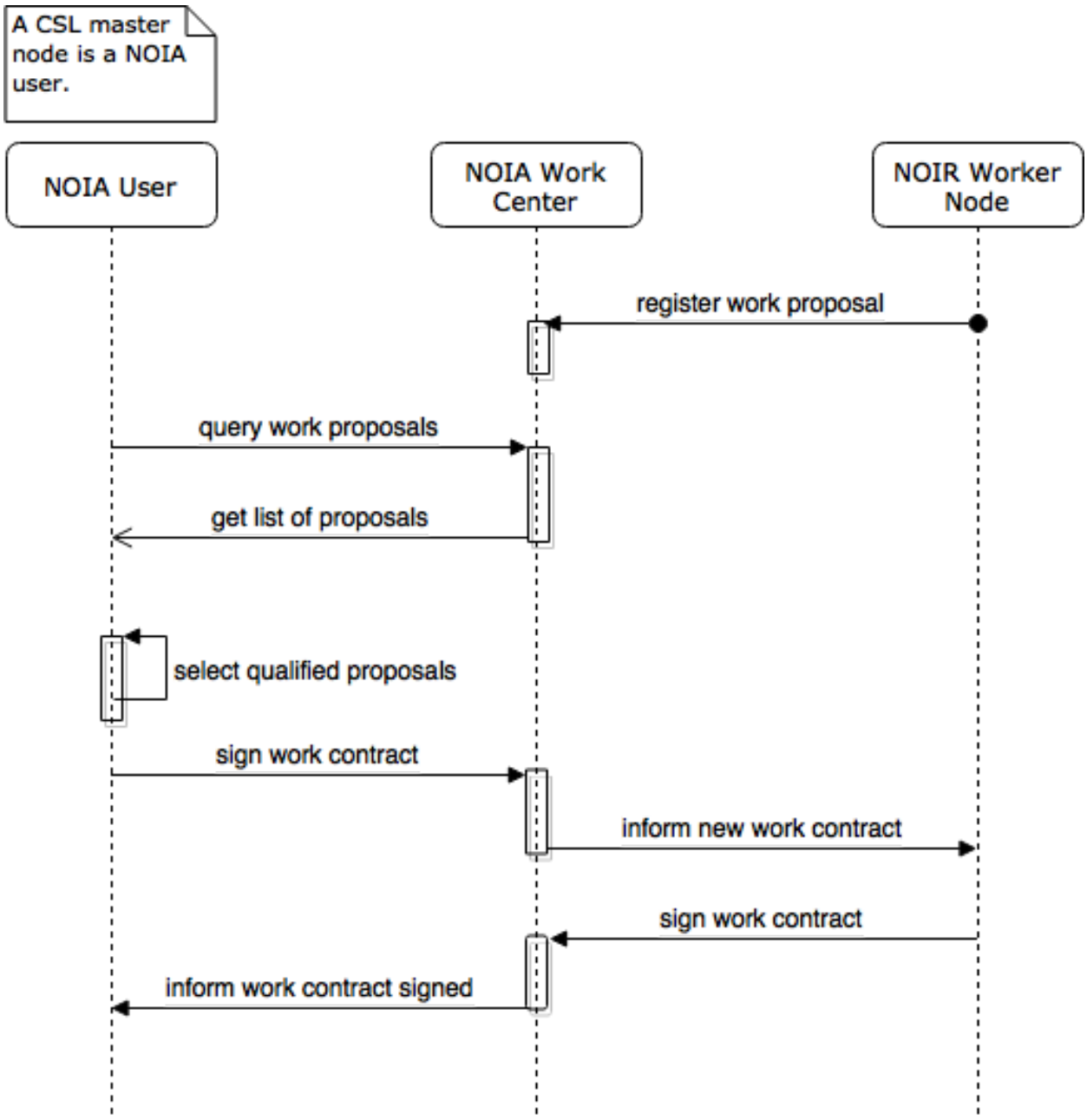
(account) and usual Ethereum wallet features such as transferring the tokens between addresses and backup / recovery of the wallet.

Since an Ethereum address alone acts as a NOIA account, all accounts are automatically transparent and anonymous. On top of this, NOIA will have an additional proprietary application layer for people to create a cloud-based account, add their NOIA accounts to it and have additional features such as monitoring or email alerts, much like many traditional cryptocurrency mining pools.

NOIA Marketplace

Hiring Process

- A NOIA user, such a as a CSL master node, uses NOIA Work Center to find suitable NOIA worker nodes to hire.
- A NOIA worker node, needs to submit work proposals along with its capabilities and availabilities to the NOIA work center.
- NOIA work center then makes these proposals discoverable to the NOIA users.
- The NOIA user then can hire the NOIA worker nodes suitable for the work required by signing work contracts with them on NOIA work center.

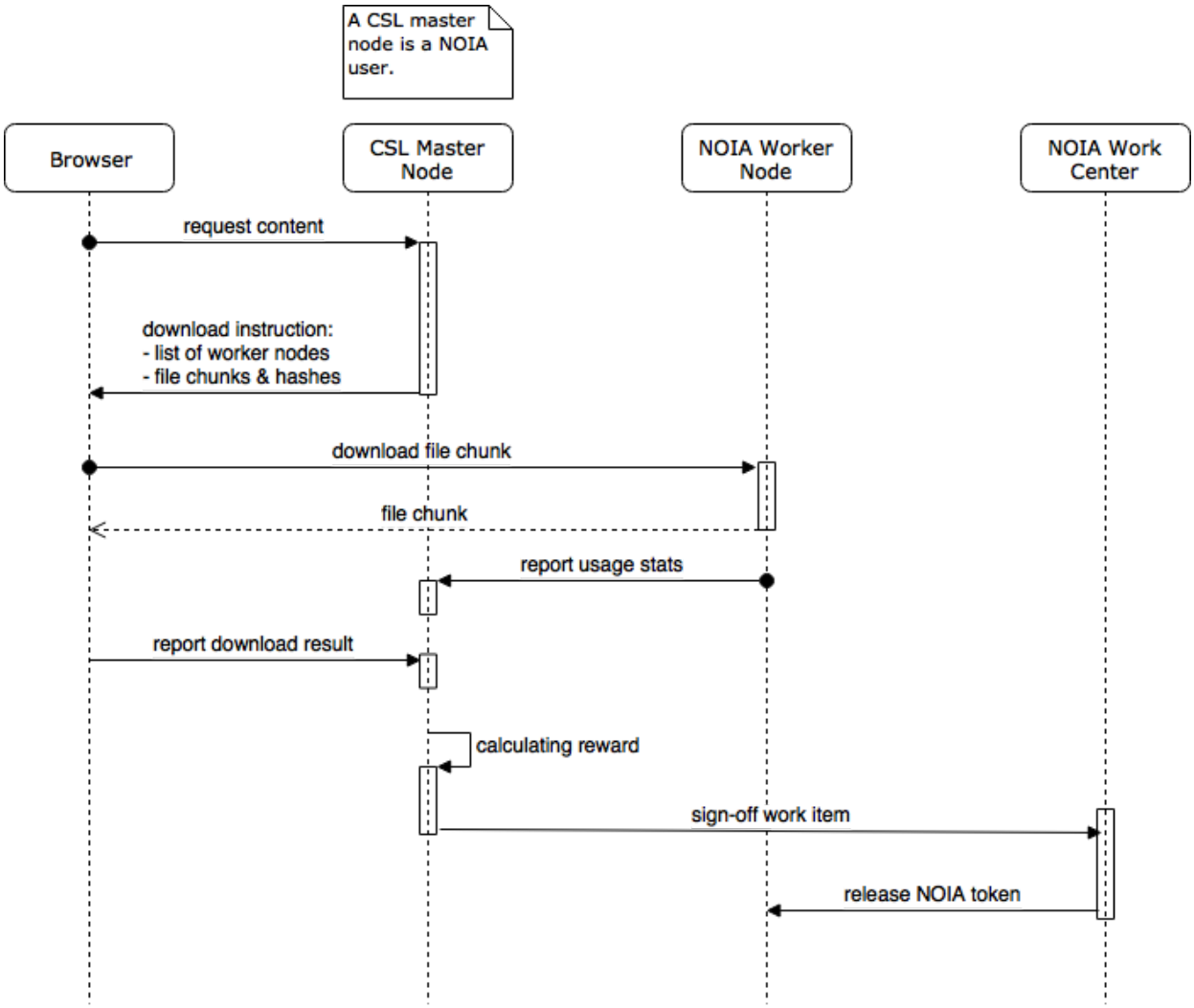


Work & Reward Process

We use CSL as an example.

- A browser gets the CSL master node selected by the CSL controller
- The browser request specific content from the CSL master node.
- The CSL master node returns a download instruction to the browser, of which a list of NOIA worker nodes is provided.
- The browser downloads file chunks from each of the NOIA worker nodes.

- When they are all done and pass the integrity check, they are reassembled into the final content.
- Upon finishing, the NOIA worker nodes send usage stats back to the CSL master node as requests for rewards.
- The Browser is advised to send download result the the CSL master node too. But they might be absent in case of the user closes the browser session too early.
- The CSL master node calculates the rewards consolidating both stats input from the NOIA worker nodes and the browser.
- The CSL master node signs-off work item smart contract at the NOIA work center. And the work item smart contract contains amount of NOIA token that is agreed by the work contract.



Smart Contracts Details

NOIA User Contract

A *NOIA user contract* contains:

- a *NOIA wallet account*,
- a Deposit.

To become a *NOIA user*, a certain amount of NOIA tokens as deposit must be prepaid. In case of a failed dispute resolution raised by a *NOIA worker node,* the credits are used to compensate the worker node.

The more *NOIA Worker nodes* a user plans to contract with, the more deposit there should be.

NOIA Worker Node Contract

A *NOIA worker node contract* contains:

- a *NOIA wallet account*,
- the public IP address of the node,
- the Performance description of the node.

It is up to the *NOIA user* to validate the authenticity of the public IP address.

The performance description consists of:

- Available bandwidth for sharing
- Available storage space for sharing
- Average uptime for sharing resources: which hours of the day, or which weekdays

NOIA Work Contract

A *NOIA work contract* contains:

- Maximum time period of the contract,
- Minimum reward amount,
- Reservation deposit.

With a NOIA work contract:

- A *NOIA user* can cancel the contract before the contract period ends, in case the *NOIA worker node* fails to deliver the performance promised in its contract.
- A *NOIA worker node* can bring a new case to the *NOIA courtroom* if it thinks the *NOIA user* is cheating instead.

- A *NOIA user* can sign off work item with NOIA token amount extracted from reservation deposit as reward to the *NOIA worker nodes* performed the work.
- A *NOIA worker node* may withhold a minimum amount of reward if the contract period ends with less reward than that amount worths of work items signed off to it.

NOIA Work Center

NOIA worker nodes present their work proposals with a “asking” price.

NOIA users can query and filter the list of work proposals available.

NOIA worker nodes and *NOIA users* can sign contracts using the hiring center, and only one active contract is allowed at a time for each *NOIA worker node*.

NOIA Courtroom

For all the work items performed by the *NOIA worker nodes*, it should be expected that those work items should be signed-off by the *NOIA user contract* with its private key:

```
NOIA Worker Node Address: 0x22222222222222222222
Data Transferred: 10TB
NOIA User Address: 0x333333333333333333333
NOIA User Signature: lk7PZhV1IXZ...
```

A *NOIA worker node* should always keep the logging of these work items, and they can be presented to the courtroom as evidences.

A required number of of *NOIA worker nodes* can form the juries and resolve the case by examining the evidence presented by the *NOIA worker*.

If the *NOIA worker node* wins the case, it will get its rightful reward back from the *NOIA user*’s deposit. Moreover, all juries will get a portion of the reward too for their jury work.

If there are collusions amongst *NOIA worker nodes* to abuse *NOIA user*’s deposit, a *NOIA user* may choose to blacklists these nodes and may publicize its suffrage to inform other *NOIA users*.

CSL Marketplace

CSL Master Node Qualification Process

For the initial version, all CSL master nodes are set-up by the NOAI team. Until all imaginable attacks are handled, the *CSL cloud controller* will not open the *CSL master nodes* market. That is not to imply these smart contracts are useless, since using the CSL subcontracts is very effective way architecturally to allow the *CSL master nodes* to gain a certain degree of autonomy. And the

autonomous *CSL master nodes* share important scaling responsibilities from the *CSL cloud controller*, so it makes the CSL network scalable and robust.

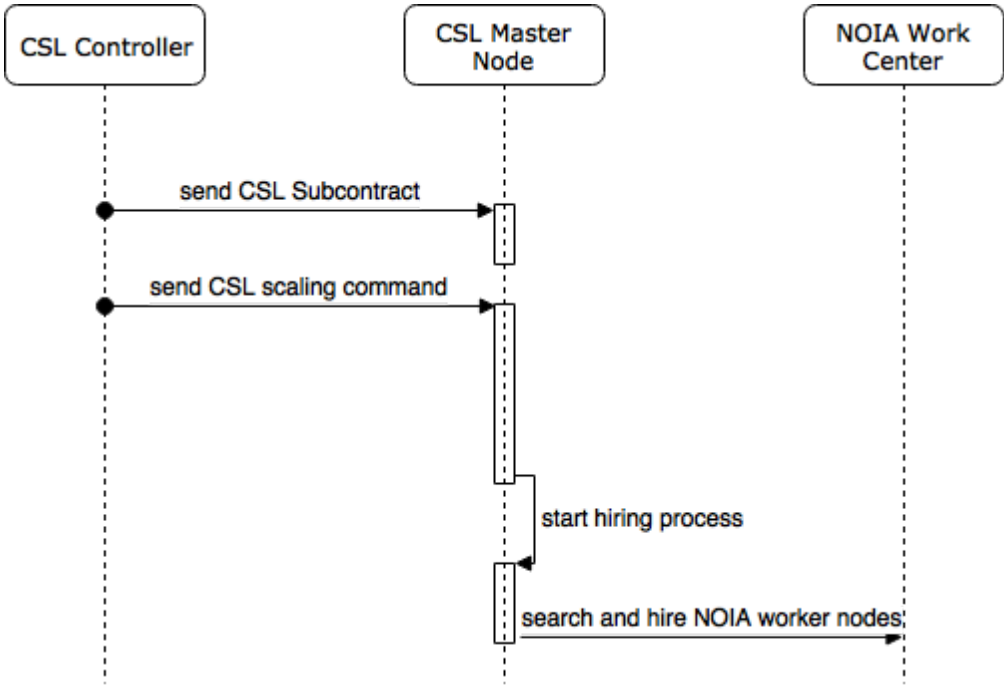
It is also critical for making qualified the CSL master node list public knowledge. Browsers may check against this information to detect unauthorized CSL master node being used.

CSL Master Node Work Distribution Process

CSL Controller offloads the direct communications with NOIA worker nodes to its master nodes. But since CSL controller manages the NOIA tokens that are contributed by its customers, it needs to distribute them to the master nodes through the smart contracts.

To do so, it creates CSL subcontract which contains a certain amount of NOIA tokens allowing one CSL master node to fully control the funds for hiring NOIA worker nodes.

After that, CSL controller issues scaling commands based on the on going content requests from browsers, machine learnings of traffic patterns and customers on-demand requests.



Smart Contracts Details

CSL Cloud Controller Contract

With the CSL cloud controller being a proprietary cloud service, this is the single instance of contract present on the blockchain.

CSL Master Node Contract

It contains:

- NOIA wallet account
- Host Address

CSL Subcontract

A bulk order from the CSL controller can be stated similarly to “Paying 10000 NOIA tokens for transferring 1000TB of data through your worker nodes”.

It contains:

- Deposit of NOIA tokens.

Each scaling instruction issued by the CSL Cloud Controller has to be linked with one CSL Subcontract.

NOIA Token economy

Worker node reward system

Worker nodes are rewarded on their resources that were utilised by the CSL during the contract period. By entering into a smart contract, a node allocates its storage and bandwidth for a certain uptime, during which it pledges to stay online. The CSL makes the best effort to utilize 100% of the resources allocated by the nodes. However, the actual awarded number of tokens may depend on the geolocation of the node and the demand of CSL services in general.

Storage / bandwidth utilization factors (F_s and F_b respectively) are the ratios of the average storage / bandwidth used over a certain time window, e.g. contract period, to the total storage / bandwidth allocated in the CSL work contract. In the similar way we derive uptime factor:

$$F_{Uptime} = Actual\ uptime\ /\ Uptime\ pledged$$

Subsequently, we can calculate an actual score and its maximum possible value – a cap of the worker node score:

$$Score_{Max} = ln(e + F_s \times Allocated\ storage) \times ln(e + F_b \times Allocated\ bandwidth)$$

$$Score_{Actual} = ln(e + F_s \times Average\ storage\ used) \times ln(e + F_b \times Average\ bandwidth\ used) \times F_{Uptime}$$

A number of tokens awarded under a specific CSL work contract is then proportional to the amount of resources that were utilized during the contract period:

$$NOIA_{Awarded} = NOIA_{Contracted} \times Score_{Actual}/Score_{Max}$$

where $e \approx 2.71828$ is Euler's number. A useful property of the logarithmic function in this case is that it is monotonically increasing with a decreasing rate. To make it work with a zero utilization of either of resources, Euler's number is added to the argument.

Scores have two main applications:

1) The proportion of the storage and bandwidth scaled by the actual uplink time results in the award calculation in smart contracts. The construction of such a reward formula allows for utilising storage or bandwidth only, as the node still gets a share of max possible reward.
2) Scores can be used to compare the worker nodes, for instance, to estimate their health. At low values, scores grows fast, thus worker nodes with even modest resources (smaller storage and lower bandwidth) are incentivized to participate in the CSL.

A presence of a particular node in CSL can be calculated by:

$$Presence_i = Score_{i,Actual} / \sum_{i \subset N} Score_{i,Actual}$$

NOIA Payment gateway

Businesses should be able to pay using FIAT currencies for CSL services provided by the NOIA network. For that purpose we will have our own payment gateway / exchange. Businesses will be able to pay using FIAT currencies, then our payment gateway will automatically convert the funds into NOIA tokens according to the latest available exchange rates or potentially using NOIA's own reserves to provide sufficient liquidity. After this, all transactions, hiring worker nodes and payouts to worker nodes, will be executed using NOIA tokens with the help of smart contracts. Payment directly with NOIA tokens will be another option, potentially with additional value benefits to encourage adoption of NOIA tokens to buy NOIA CSL services.

References

Abreu, R. and Cardoso, N. 2014. An Efficient Distributed Algorithm for Computing Minimal Hitting Sets. *Proceedings of the 25th International Workshop on Principles of Diagnosis (DX'14)*.

Chen, M., Chowdhury, R.A., Ramachandran, V., Roche, D.L. and Tong, L. 2007. Priority Queues and Dijkstra's Algorithm. *UTCS Technical Report TR-07-54*.