Internet of People

Software Requirements Specification

Version 0.7 - Nov 2016

Authors

Luis Fernando Molina

Contributors

David Matousek, Wigy, Daniel Roka, Matias Furszyfer, Rodrigo Acosta, Miao ZhiCheng, Istvan

This document describes the requirements for a complete IoP implementation.

Introduction

The IoP is a concept introduced by the Internet of People white paper. We assume in this document that you are familiar with this document and the ideas introduced there.

Purpose

The main purpose of this system is to create a public network of people profiles that anyone can use to find, connect and interact with others without needing to go through the web or private networks.

A secondary purpose is to enable people to build a cross-profile and cross-industry online reputation to be used for any purpose end users see fit.

All of this with an economic incentive for the operators of each type of network node and server.

Definitions

These are a few definitions that make the following content unambiguous:

- **IoP Protocol:** It is the set of interfaces that defines the interaction between IoP client apps, IoP nodes and IoP Servers. It also defines the interaction between nodes / servers themselves.
- **IoP Node:** Instance of one of the IoP nodes: IoP Unstructured Network (UNN), IoP Content Address Network Node (CAN), IoP Location Based Network Node (LOC), IoP Latency Base Network Node (LAT).
- **IoP Server:** Instance of one of the IoP servers: IoP Token Server, IoP Profile Server, IoP Proximity Server, IoP Reputation Server.
- **IoP Client App:** or IoP App is any software that connects to and uses the IoP network (any IoP network). To do so it uses the IoP Protocol. Software components of the IoP p2p networks are not considered IoP Clients in this although they might consume services from each other.
- **Node Identity:** It is a private & public key pair.
- **Node Network Id:** It is the key assigned by the network to a node to participate on the routing tables that help other nodes or clients to find it by id.
- **Node Profile:** Node pub key + (IP + Port) + Node Network Id + Node Location.
- **Location:** Latitude + Longitude.
- **Profile:** It refers in general to end users profiles.
- **Profile Types:** There are different types of end user profiles, as for example: chat profile, artist profile, merchant profile, etc. IoP does not limits the profile types that can exists. Apps might define a new profile type or they might stick to a profile type already existing. IoP do defines a standard set of Profile Types for apps that want to interoperate between each other. This standard set together with the specific messaging associated with that profile type is defined by the IoP Consortium.

- **Personal Profile:** A Personal Profile is all the information end users define for a certain profile. This can be anything at all. This information is divided in two main parts:
    - **Network Profile:** It is the part of the profile that is uploaded to their profile server and it is 100% public. This information is again splitted in two:
        - **Home Profile:** It is the part of the information that is kept only at their profile server. This could include for example what a person is selling, doing, like or a similar information.
        - **Shared Profile:** It is information shared with neighboring nodes in order to be easily found. A common example is an alias, but there is no restriction in what it can be. It is only constrained by the space reserved for this.
    - **Device Profile:** It is the part of the profile that is kept at the user's own device and thus only accessible by whoever the user provides it access to. Usually this happens after having a relationship of some kind.
- **Profile Relationships:** Relationships established between profile owners. IoP does not define relationship types nor does it store that information on its networks.
- **Relationship Card:** It is a data structure that contains a prove that a certain entity has a relationship with another entity. When a relationship is established entity B signs the content of the data structure that includes the public keys of entities A and B as well as the relationship type and an expiration date. For example, Bob signs a card that says that Alice and Bob are friends and give that card to Alice. Then Alice can use that card to prove she is Bob's friend to anyone until the card is expired. This card might be used by Alice to endorse Bob on how such a good friend he is, as well as for example to receive notifications from Bob's profile server when Bob is online.
- **Application Service:** It is an application level communication channel between two remote devices. There can be many different application services and each one has its own communication protocol. IoP does not define application services, client apps using IoP do. Each client informs its Profile Server which application services are available on its end. Once the application services are checked in, other devices can call this client through any of them.
- **Full Network Map [FNM]:** It is a table that contains all nodes profiles existing in the network. LOC nodes running with the default configuration usually don't maintain a Full Network Map, just a part of it. Only at the early days of the network, when the amount of nodes is low, the Full Network Map could be found at any node.
- **Local Network Map [LNM]:** It is a part of the Full Network Map that is different for each node and it reflects a node's knowledge about the network. Nodes to be part of the Local Network Map of a node are selected by the Local Network Map Algorithm described below in this document.

References

To understand this document you are required to read:

- **IoP White Paper:** [unpublished draft] https://docs.google.com/document/d/1fwXOCfX0zfJv-MloeVAsVjS_JEpfdIXHUObMBijLrJ8/

Overall Description

The IoP Network is a mission critical infrastructure and the highest level requirements are derived from the following list of properties it needs to have:

- **High Availability:** The entire network must never be down. IoP nodes run in an uncontrolled environment - i.e. Node Operators' computers and servers. It is expected a wide range of diversity on the hardware used, eliminating the possibility that the hardware can end up being a single point of failure. However a flaw on the software implementation could halt the whole network in certain scenarios. To fulfill the requirement that the entire network must never be down more than one independent implementation of the IoP protocol is needed.

- **High Performance:** Using the IoP should be as efficient as using a regular cloud / web service for most use cases. Nodes should be designed in a way that allows their operators to maximize their profit, but providing a quality service only. Nodes should only serve as many users as they can handle with a quality service. To reach this level of self awareness nodes will need extra intelligence that can be added over time.

- **High Scalability:** The IoP should scale globally and be able to handle the whole world population. The network must scale to any number of nodes, meaning that the algorithms and relationships between nodes can not include requirements of resources that grow linearly with the total number of nodes of the network.

- **Highly Secure:** The network should be resistant to a wide range of possible attacks. Attack vectors must be identified and prioritized.

- **Minimum Maintenance Risk:** The risk of upgrading a node software should be the minimum possible and contained in a way that the risk of a change in some functionality can not be spread across the whole system. The highest level division will be splitting the services the network provide in independent p2p networks where the risk of a change in one of them can never spread to the other networks unless the IoP protocol itself is changed. Modularity inside each network implementation is also highly recommended.

Besides these higher level requirements, IoP nodes must be:

- **Free to Join:** Any node should be able to join the network at anytime. In the context of each individual p2p network this means different things. The network itself must not be affected by a new node joining and should be prepared for that.

- **Free to Leave:** Any node should be able to leave the network without causing a major damage to the service quality the network is providing. In each p2p network the procedures to handle a node leaving might be different. The service provided by the network should be impacted the least possible by the event of a node going offline.

The IoP protocol will evolve with time as well as the individual implementations. As nobody can control which version is being run both of the protocol and the implementations, we should expect multiple versions to run at the same time. For this reason the system must be:

- **Multi-Protocol-Version:** Nodes must be aware that they must have relationships only with nodes with a compatible version of the IoP protocol. Higher protocol versions must be ignored. Intelligent nodes could sense if there is a trend towards moving to a higher version and suggest its operator to upgrade.

- **Multi-Client-Version:** Clients might be running different version of the IoP protocol. Nodes must be prepared for this fact and support more than the latest version.

Product Perspective

We aim to create nine product families:

- **IoP Token Server and Related Software:** In this family we have the IoP Token Server and Network Node and a basic CPU miner.

    - **IoP Token Server:** It is the software that runs the IoP blockchain. It also works as an IoP wallet and can give block information to miners.. This software allows their users to earn IoP tokens given as mining rewards and as transaction fees when IoP tokens are transferred from one party to another. The roam map of this component is tricky since at some point it is going to be splitted into different components. The software early release lifecycle for this product is:

        - **Close Beta Release:** Same codebase as bitcoin, same hashing algorithm and PoW. Mining restricted to trusted community local chapters. Wallet usage open to anyone. The main differences with bitcoin code base are:

            - A list of trusted community members public keys is read from the blockchain introduced there as outgoing transactions from a special control address.

            - New mined blocks are accepted if the coinbase transaction includes the coinbase transaction signature produced by the private keys of the listed public keys.

            - Amount of tokens per block reward during this stage is reduced.

            - A piece of code is added to generate the pre-mining at the second block.

            - A external program is used to distribute the pre-mined coins with a lock constraint in the future.

        - **Voting System:** A governance system is introduced in order to allow contributors to be paid by newly issued tokens.

        - **Chapter Management System:** A system to manage local chapters is implemented at this component as a database layer.

        - **Release:** Stable final version. Anyone is allowed to mine at this stage. The normal rate of token issuing is re-established.

- **IoP Profile Server and Related Networks:** In this family we have the IoP Profile Server and two p2p networks which the server depends on: the Location Based Network, and the Content Address Network.
    - **IoP Profile Server:** It is the software product that allows its users to earn IoP tokens by being the Profile Server of IoP end users. The software release lifecycle for this product is:
        - **Standalone Server Releases:** This release allows a server to serve the Client to Server use cases described below. There is no network, just one single server running. At this point, the server can be tested serving existing IoP apps. The expected releases for this phase include:
            - **Hosting Profiles Release:** Functionality related to hosting profiles is added. No payments / sell of services is implemented at this stage. This functionality has already been implemented.
            - **Connecting Devices Release:** Functionality to allow devices to connect to each other through the server is added. This functionality has already been implemented.
            - **Hosted Profiles Queries Release:** Functionality related to querying the profile information hosted is added here. This functionality has already been implemented.
            - **Shared Profiles Queries Release:** Functionality related to querying the shared profile information available of the neighborhood. This functionality has already been implemented.
            - **Relationship Cards Release:** This allows profile owners to link their profiles between each other, or different profile owner to publicly express relationship between profiles. This functionality has already been implemented.
            - **Integration with LOC (Location Based Network):** This allows the server to be part of the location based network and be aware of its neighbourhood. This is being implemented right now.
            - **Integration with CAN (Content Address Network):** This allows the server to to be part of the content address network and end users to find profiles by the public key of the owner. Implementation is pending.
            - **Backup Nodes Releases:** This includes the functionality for a node to take the role of a backup profile server of a certain profile.
            - **Payments Releases:** These releases includes what is necessary for nodes to charge IoP tokens for their services. List of releases to be determined.

- **Admin Interface Releases:** This includes the functionality to manage the node remotely from an external GUI apps. List of releases to be determined.

- **Notifications Release:** Functionality related to notifying clients when hosted profile info is changed. Implementation is pending.

  - **Location Based Network Node:** This node creates a location based p2p network. It is initially needed by the Profile Server but it will also give its service to other servers like the Proximity Server.

    - **Location Based Network Node Releases:** This includes the Node to Node Use Cases described below at the LOC section of this document. The expected releases for this component include:

      - **Unrelated Nodes Release:** Functionality provided for unrelated nodes is added. This functionality has already been implemented.

      - **Colleague Nodes Release:** Functionality provided for colleagues nodes is added. This functionality has already been implemented.

      - **Neighboring Nodes Release:** Functionality provided for servicing neighboring nodes is added. This functionality has already been implemented.

      - **Routing by Location Release:** Functionality to enable routing assistance by location is added.

- **IoP Proximity Server and Network:** This family have only the IoP Proximity Server product.

  - **IoP Proximity Server:** It is the software product that allows its users to be part of the IoP Proximity Network and to earn IoP tokens by allowing IoP end users to find themselves while they are at the same location.

- **IoP Reputation Server and Network:** This family have only the IoP Reputation Server product.

  - **IoP Reputation Server:** It is the software needed to be part of the IoP Reputation Network. It allows its users to earn IoP tokens by providing reputation services to IoP end users.

- **IoP Minter Network:** This family has only one product.

  - **IoP Minter Server:** It is the software that audits all other full nodes to determine who is a Real Full Node and may become a IoP Minter. To do so, it runs a set of tests that according to a score, will determine if a Full Node can be upgraded to the rank necessary to mint IoP tokens.

System Interfaces

The IoP System does not have dependencies on other external systems.

User Interfaces

The IoP System does not have a user interface. Nodes and Servers run according to what is configured at configuration files. Each software component provides a management API that enables user interfaces for management to be developed.

Hardware Interfaces

The IoP System hardware requirements are very basic and straightforward: hard drive space, cpu, memory and networking. Internet connectivity needs to be available for nodes and servers to run. Node users are able to configure the maximum amount of each type of resource they allow their node to consume.

Software Interfaces

This is the list of software dependencies:

- For the C# implementation only .NET core is needed as a dependency.
- For the Java implementation the Java Runtime is needed as a dependency.
- For the Rust implementation no dependencies are needed.

Communication Interfaces

The communication between IoP clients and nodes and between IoP nodes is via TCP/IP.

Design Constraints

Operations

The system can be monitored through an API that returns statistical information. This can be used to diagnose and prevent potential problems. Statistics can also be used as an input on discussions on how to refine and fine tune IoP standards.

Specific Requirements

To introduce this section we describe the high level services each network is going to provide.

Token System Components Requirements

Status

Currently under development by Rodrigo Acosta.

Version 2.0 already released.
Next release scheduled for late November 2016 : Voting System.

Worked at:

- https://github.com/Fermat-ORG/iop-token
- https://github.com/Fermat-ORG/iop-token-distributor
- https://github.com/Fermat-ORG/iop-mining-admin-tool
- https://github.com/Fermat-ORG/iop-token-redeemer
- https://github.com/Fermat-ORG/blockchainj
- https://github.com/Fermat-ORG/iop-watchonlywallet-api

Working at:

- IoP Token Voting System

Introduction

Premined tokens distribution

Block #1 rewarded 2.1M tokens as premined coins to developers and stakeholders who contributed on an early stage. The IoP-Token-Distributor and IoP-token-Redeemer applications were developed to distribute and be able to redeem these coins.

IoP-Token-Distributor: as the name implies, this application was used to distribute the premined tokens to the correct people. Is a Java console application that takes the Token internal spreadsheet as input and generates and broadcast IoP transaction as output. From the spreadsheet, it takes the destination address and values to distribute, and any time constraint if existed.

The admin of the premined tokens, the person that own the private key that mined block #1, executed this application to perform the distribution on several stages until completed.

70% of the premined tokens was available to be spent immediately, so most of the generated transactions where standard in the form:

Premined Input -> Multiple outputs in the form (OP_DUP OP_HASH160 PubKeyHash OP_EQUALVERIFY OP_CHECKSIG)

-> change back to admin key

For example:

http://explorer.fermat.community/tx/9544b8fff910be5b2cf11008bfd848ad5f64c3b362abef0816ba0cfad49094c6

30% of the coins have time constraints to be spend after 6 months, 1, 2, 3, 4 and 5 years. In this case, the application generated P2SH transactions in the following form:

Premined Input -> Multiple P2SH in the form (OP_HASH160 RedeemScriptHash OP_EQUAL)

For example:

http://explorer.fermat.community/tx/b48c0bed29881f34c8539bb9e4529a14883806f2b04fddae7ca597ab6b427f97

The Redeem Script created by the application uses the Op code OP_CHECKLOCKTIMEVERIFY to enforce a time constraint in the future to be able to spend those coins. The ScriptPubKey generated has the following form:

EpochTime OP_CHECKLOCKTIMEVERIFY OP_DROP OP_DUP OP_HASH160 AddressHash OP_EQUALVERIFY OP_CHECKSIG

For example:

1493408180 OP_CHECKLOCKTIMEVERIFY OP_DROP OP_DUP OP_HASH160 3ecb7f12ec91b7c4861efb97f3a814f93532e6d2 OP_EQUALVERIFY OP_CHECKSIG

This redeem script that was generated automatically by the distribution application will allow the right private key holder use the coins sent after Fri, 28 Apr 2017. Anyone submitting a transaction after that date, complying with the rules of BIP 65 and signing the transaction with the appropriate private key will be able to claim those coins.

IoP-token-Redeemer: In order to be able to redeem the time constraint P2SH transaction explained just before, a Java console application was created to easily generate the transaction. In other words, the IoP-token-distributor application presented a puzzle, the IoP-token-redeemer presents the solution to that puzzle.

As explained in the Readme on github, this application takes more inputs in order to be able to find the correct transaction and P2SH output that we want to claim. The goal is to create a “final” transaction which includes a ScriptSig with the following data to include into the IoP Core stack:

RedeemScript → Address → Signature

This transactions specifies a LockTime equal to the one provided on the original transaction and an nSequence value equal to 0, so it can be considered final. This restriction won't’ allow us to submit a transaction before the specified time has come to past.

The RedeemScript pushed onto the stack will resolve the original ScriptPubKey of the transaction:

OP_HASH160 HashRedeemScript OP_EQUAL

The Locktime of the transaction will resolve the BIP 65 portion:

EpochTime OP_CHECKLOCKTIMEVERIFY

And the Address and Signature will resolve the rest of the redeem script

OP_DROP OP_DUP OP_HASH160 AddressHash OP_EQUALVERIFY OP_CHECKSIG

Being able to comply with all the rules of the ScriptSig of the original P2SH, the owner of the private key that signed this new transaction is able to spend the tokens.

Blockchainj

Blockchainj is a fork from the great bitcoin java library bitcoinj. The blockchainj library maintains all the bitcoin functionality, but it also expands it to the Internet of People blockchain.

Just as it was expanded to support IoP transactions, more blockchains can be added the same.

For bitcoin, the following classes define the parameters and properties of the bitcoin network:

- BTC_MainNetParams
- BTC_TestNet3Params
- BTC_RegTestParams

In a similar way, IoP defines its parameters using the static methods of the following classes:

- IoP_MainNetParams
- IoP_TestNet3Params
- IoP_RegTestParams

Since IoP has different validations and properties than bitcoin, depending on the selected network, the correct validations are executed. For example:

```
if (params.getSupportedBlockchain() == SupportedBlockchain.BITCOIN)

  scriptsigMaxSize = 100;


if (params.getSupportedBlockchain() == SupportedBlockchain.INTERNET_OF_PEOPLE)

  scriptsigMaxSize = 220;
```

Since in IoP we use the ScriptSig of the coinbase transaction to validate that only whitelisted miners can mine, the size of the scriptSig script is bigger than in bitcoin.

IoP Watch-Only API

This api creates a watch only wallet that contains only public keys. It is suitable to be used on non-secure servers and its job is to notify of incoming coins to any of the addresses the wallet monitors.

Using blockchainj and creating an SPV client, the wallet creates a key hierarchy of public keys only and connects to IoP full nodes to retrieve block headers.

On its initial version, the wallet does not derive new public keys from an extended public key. Since the IoP Core wallet has a basic HD key hierarchy which derives only hardened keys, there is no need at this point to create a watch-only wallet from its own seed to be able to derive new addresses.

When the IoP Core wallet functionality is expanded to be able to derive non hardened keys and generates an extended public key, a new version of this API will use that extended public key to derive new public keys without using private keys.

The final version of this api will be able to derive the same addresses that the IoP wallet without using any private key.

IoP-token

Rodrigo: To be continued...

Token Issuing

Economics

The IoP Token System is capped at 21M tokens to ever be issued. This number is arbitrary and inherited from bitcoin. Any number would be arbitrary, so we prefer to have the same number as bitcoin to facilitate the comprehension of the IoP economy by people that already know bitcoin.

Like bitcoin, IoP also has halving events or a decreasing supply with the same rules. The amount of tokens per block is variable and capped at 2. One of the tokens per block goes to miners / minters and the other is issued only when there are approved *Contribution Contracts*. The final target block frequency is every 2 minutes. This means that at full speed, the IoP issuing will run at a rate of maximum 1 IoP token per minute.

Mining IoP tokens will be done in the most decentralized and resilience way possible. In the very long term securing the network by mining will be done by different networks that we will build over time. The mining will evolve until it's final state where a block is going to be mined by one network, the next block by a second network and so on, including end users as one of the possible networks. Current network ideas includes these ones:

1. Fermat Chapter Network → We are building this one today.
2. Fermat Research Network
3. Fermat Business Network
4. Fermat End Users Network

Block Frequency Schedule

The project starts with a 10 minutes block frequency and will be reduced to 2 minutes in several steps tied to these milestones:

Release	Target Block Frequency

IoP Blockchain	10 minutes
Voting System	9 minutes
LOC & CAN Network	8 minutes
Profile Server	7 minutes
Chapter App	6 minutes
STUN & TURN Servers + LAT Network	5 minutes
Proximity Server	4 minutes
Reputation Server	3 minutes
Minting & Token Servers + UNN	2 minutes

The final block frequency will be a block every 2 minutes, and the issuance will be capped at 2 token per block maximum. 1 for miners / minters, 1 for paying *Contribution Contracts* voted by token holders.

IoP / Bitcoin Economics Comparison

IoP will be issuing once all components are released 10 tokens every 10 minutes (2 every 2 minutes). Both systems have the same halving rules for the issuing. Bitcoin started with 50 tokens every 10 minutes. That means that IoP issues at a rate 5 times less than bitcoin. Bitcoin issuing progression will last for around 120 years, in contrast IoP issuing progression will last around 5 times more. (Note 10% IoP were pre-mined and distributed to early contributors).

Contribution Contracts

The IoP blockchain has an embedded voting system similar to the one implemented in DASH. The main difference with DASH is that they allow what they call *master nodes* to vote, while we allow any token holder. Our approach is more inclusive, since you don't need to be a geek to vote. Even investors with little or no technical knowledge can cast votes through a custom IoP Voting App.

Contribution contracts initially are proposals made by any individual or entity to token holders (community). These contracts include precise information regarding what contractors would do, when they will start and finish, what are the deliverables and how much they will charge.

Voting System

Anyone is encouraged to submit proposals. There will be an specialized IoP App for IoP Contributors. These proposals are later discussed at an open forum. Token holders are later able to vote with their Voting App. So how will this work in detail?

IoP Contributor App

1. Anyone can download the IoP Contributor App (mobile app).

2. This App is a kind of specialized IoP wallet, since it can receive IoP tokens and use them.

3. Once funded, end users can create a profile that will be uploaded to a Profile Server. The profile can be of a public real person or someone that want to remain anonymous.

4. Before submitting, the contributor must prove to be invested in the project. An amount of Contribution Contract Collateral (CCC initially of 1,000 IoP) must be deposited at one of his own accounts of the internal wallet of the APP, and be kept there during the whole execution of the contract in order to be considered valid. Removing this amount is also a way to unilaterally withdraw from the Contribution Contract.

5. Then they are ready to submit a proposal. Submitting is not free, since it must be recorded at the blockchain. The submitted transaction must comply with the IoP Contribution Contract and Voting protocol specifications described below.

6. The mining fee paid by the submitter is later used to order the proposal at the screen of the IoP Voting app. The higher the fee paid, the higher in the list to be considered for voting. There is a minimum fee of initially 1 IoP to prevent spam. This minimum value will later be recalculated automatically.

7. Several beneficiaries can be specified at the Contribution Contract. All of them are going to be recorded in a single transaction at the blockchain. An API will be added to receive new Contribution Contracts and record them on the blockchain. In this way a single contract can be used to pay a team of people, each one receiving some arbitrary user defined part of the total contract amount.

8. The Contribution Contract data is posted to a web forum where the discussion is going to happen.

9. As this App is an specialized wallet at the same time, it allows end users to withdraw their funds from the app. Also back up the internal wallet if they want.

IoP Voting App

1. Anyone can download the IoP Voting App (mobile app).

2. This app is also like a specialized wallet that can receive IoP tokens to be used as voting power.

3. They can see their available balance on the main screen.

4. On a secondary screen they see the Contribution Contracts available for voting ordered by the mining fee each contributor payed.

5. A third screen will be the forum discussion, with possibilities to post new comments from the APP.

6. On a fourth screen they see the Contribution Contracts which a vote from this user has already been casted and are still open, meaning they can change their vote.

7. Finally on another screen they can see their voting history, meaning Contracts already closed.

8. They can enter into the detail of a Contract and see the following information:
    a. Title
    b. Sub Title
    c. Category (Development, Marketing, PR, etc)
    d. Short Description (to be placed at the Contract Detail screen)
    e. Starting Block Height
    f. Ending Block Height
    g. Tokens per Block to be Paid.
    h. Web Forum Post Id
    i. Deliverables List (Name, Link, Status) Status might be updated by the Contractor during the process.
9. They can select any amount below their current available balance and write it on a Contract detail screen before casting their vote.
10. They will have a switch placed by default in the middle, between YES and NO. Swiping the switch to the left they will vote YES and to the right NO. After doing so, the wallet will wait 10 seconds doing nothing just in case the user wants to cancel the operation. After that it will proceed to vote.
11. Internally the app will move the amount selected to a new account of the same internal wallet paying 1% mining fee. The balance on that account must be untouched for the time the Contract is active, in order to be considered a valid vote. If funds are removed or more funds added the vote will be ignored by the voting engine implemented as a new rule at the blockchain level. This 1% fee is necessary to go with the saying "put your money where you put your mouth".
12. During the lifecycle of the contract, the vote can be withdrawn by moving the slider to the center. This will generate a transaction that moves the funds in the previous account to a new account of the wallet, adding this funds to the available balance. Only the default mining fee is used for this.
13. Naturally the IoP Voting App will consider the funds in the vote address as available balance once a Contract enter into Closed status (meaning that it was fully executed).
14. The app must allow the user to withdraw fund from the available balance to an external account. Also to make an internal wallet backup.

The Voting Engine

1. The blockchain voting engine will consider each token at the balance for a YES vote at the rate 1 Token = 1 YES vote. For NO votes, it will consider 1 Token = 5 NO votes. Token fractions are considered. In other words, NO balance is multiplied by the factor 5. This is to prevent big stakeholders to game the voting system.

2. There is no limit regarding how much voting power to use. On the lower range, the limit is 1 IoP-toshi. On the upper range, the limit is the amount of token the voter has available at the moment of casting his vote.

3. The Contract specifies the amount of tokens to be paid per block, the initial block and the final block. At each block before issuing the new tokens the engine verifies all the votes related to that contract and computes if it must be payed or not. To be considered approved, at each block the computation must comply with YES votes > NO votes.

4. We assume that voters can change their mind at any time and remove their YES vote, or even cast a new NO vote. Also new voters can enter anytime voting NO. We will not accept new YES votes after First Block - 1000. This is to prevent the beneficiaries to vote themselves at the last minute or during the execution of the contract. Once the blockchain block height equals First Block - 1000 that contract fate is sealed. If it did reach the condition YES > NO by them it will be considered an *Approved Contract,* if not it will be considered a *Not Approved Contract*. During the next 1000 blocks its status will be *Queued for Execution.* Once the block height is >= First Block the contract is *In Execution*. Once the block height is > Final Block the status is *Executed.* If NO votes truncates the lifecycle of the contract, depending how far the contract was, it can end in *Dequeued* or *Execution Cancelled* status. The IoP Voting app might use these status for notification to voters.

5. Contracts are processed by the order they were submitted. That means that if the processing engine reaches the limit of 1 token in a certain block (or is very close to that limit) having already paid the first contracts in its list, and the next contract in the list would pass that limit, it will ignore it and the rest of the contracts after that will be ignored as well. To avoid this erratic behaviour, reference IoP Voting apps will prevent users to vote for proposals that don't have enough room to be payed. The engine anyway will take into consideration this rule because other voting apps might be developed in the future that overcomes this restriction.

6. We will impose a limit to the amount of the Contribution Contracts and to the amount of blocks needed to pay for the contract. This is to prevent miss use and encourage short term contracts whose deliverables can also be verified in the short term. The amount limit will be fixed in 0.1 token per block. The amount of paying blocks will be fixed to target a time period of one month. This means that the amount of blocks will depend on the Block Frequency Schedule.

7. The engine will not consider valid Contracts those that require payments before *n* blocks since the Contract was recorded. This is to prevent an attacker to create a Contract vote it YES and immediately collect the tokens. *N* is tied to the Block Frequency Schedule. Its initial value is n = 2016 (2 weeks at a 10 minutes per block frequency). Its final value is n = 2016 * 5 (2 weeks at a 2 minutes per block frequency). The *n* value during the phases in between is also adjusted to target a 2 weeks waiting period.

We believe that the previous rules will lead to a democratic environment where several things are prevented:

- Big token holders should use only part of their power since the funds movements prevent them to vote with tokens in cold storage.

- Votes are going to be casted only by people willing to pay 1% in fees. Meaning they really have an understanding of what they are voting for. In some cases it might be the same beneficiary of the Contract, leaving the opportunity for others to oppose with a factor 5 leverage over NO votes

Contribution Contract Collateral

This CCC is required in order to prevent the voting system to be spammed with irrelevant proposals. The system starts with a value of 1,000 IoP and this value is then recalculated with the same frequency and the same process that computes the new difficulty of the network.

In the case of Network Difficulty, the target is in Bitcoin a block every 10 minutes. So the difficulty is adjusted in order to pursue that target. In our case it can be a little bit more complicated:

- **CC / Active Contributors Ratio:** A low % of CC proposals in relation to active contributors could mean the CCC is too high.
- **Votes / Active Voters Ratio:** A low % of votes for proposals in relation to active voters could mean the CCC is too low and thus too many CC proposals to review.
- **Rejected CC Proposals %:** A high % of rejected proposals could mean that the CCC is too low and not serious CC proposals are submitted.

The calculation of the next CCC uses the previous value and multiplies it by a factor that depends of the 3 previous analyzed elements:

CCC = f(x, y z)

Protocol specifications

Contribution Contract definition

Inputs

- No restrictions on amount or order of inputs.
- Sum of coins on inputs must be greater than 1000 IoPs

OP_Return:

- Op_Return output is mandatory with the following format:

Field	Description	Size
Contract tag	A tag indicating that this transaction is a IoP Contribution Contract transaction. It is always 0x4343.	2 bytes
Version number	The major revision number of the Contribution Contract Protocol. For this version, it is 1 (0x0100).	2 bytes

Block Start height	The start block height of the contract. Defined as: BlockStart = current Height + 1000 + n Max: 119960 = 6 months window (0x1D498)	3 bytes
Block End height	The end block of the contract, in the form EndBlock = StartBlock + n. Max: 20160 blocks (4 weeks) (0x4EC0).	2 bytes
Block reward	Amount of coins each block will generate for this contract. Max is 10,000,000 IoP-toshis = 0.1 IoP (0x989680)	3 bytes
Contract metadata hash	The hash 256 (SHA256) of the contract metadata. See Contract Metadata	32 bytes
ForumId	Id of the forum discussion	4 bytes

OP_Return Max is 80 bytes, which leaves yet 35 bytes free.

Example

0x4343010051 2C4C4B40[Metadata Hash…]

5: Block Start height

Current Height is 8282 blocks so contract will start at height:

BlockStart = current Height + 1000 + n

BlockStart = 8282 + 1000 + 5

BlockStart = 9287

12C: Block End height = 300

EndBlock = StartBlock + n

EndBlock = 9287 + 300

EndBlock = 9587

4C4B40 = Block reward = 5000000 = 0.05 IoP / block

Meaning the contract reward is: 0.05 * 300 blocks = 15 IoPs

Outputs

Depending of the index position of the Output regarding the OP_Return output, it will be used differently:

Outputs before OP_Return must be:

- Output of freeze address with 1000 IoP value.
- Change output returning any exceeding coins from the inputs.

Outputs after OP_Return are considered Contract Beneficiaries.

- The value of each output represents the block reward of the beneficiary

Example:

Input	Output
Index 0: 40.8 IoP	Output 0: Freeze Address: 1000 IoP
Index 1: 300 IoP	Ouput 1: Change Address 29 IoP

Index 2: 500 IoP	Output 2: OP_return with BlockReward of 0.05
Index 3: 190 IoP	Output 3: Beneficiary 1 0.03 IoP
	Output 4: Beneficiary 2 0.02 IoP

The fee for this transaction will be:

Sum Inputs: 1030.8 IoP

-

Sum Outputs: 1029.05 IoP

Fee: 1.75 IoP

- Sum of beneficiary values must be equal to BlockReward field.

Contract Metadata

The Contract metadata hash of the OP_return is the Hash256 result of the following metadata:

* Id	Unique Identified
* Title	String - Contract Title
SubTitle	String - Contract Subtitle
Body	String - Contract body
Category	String
* StartBlock	Int
* EndBlock	int
* BlockReward	long
Web Forum Post id	link?

Deliverable List	{key, value}

The Metadata can be actually anything, we need to discuss if we want to define parameters of what to include or not.

Contribution Contract States

SUBMITTED: Transaction confirmed on blockchain. No votes yet.

APPROVED: YES > NO. Current height > (BlockStart + 1000 blocks).

NOT_APPROVED: NO > YES. Current height > (BlockStart + 1000 blocks)

QUEUED_FOR_EXECUTION: YES > NO. Current height < (BlockStart + 1000 blocks).

DEQUEUED: NO > YES. Current height < (BlockStart + 1000 blocks).

IN_EXECUTION: YES > NO. Current height > BlockStart

EXECUTION_CANCELLED: NO > YES. Current height > BlockStart

EXECUTED: YES > NO. Current height > BlockEnd

Voting System definition

Inputs

- No specifications for inputs

OP_Return

- Op_Return output is mandatory with the following format:

Field	Description	Size
Voting tag	A tag indicating that this transaction is a IoP Voting transaction. It is always 0x564f54.	3 bytes
Voting Power	The voter decision. 1 = Yes, 0 = NO	1 byte
Genesis Transaction	Sha256 hash of the transaction that originated the contract	32 bytes

Outputs

- Output index before OP_Return output is considered the freeze address Output.
- Outputs after Op_Return are considered change address and ignored.
- Freeze address output value is the amount of votes the user is giving to Yes or No.

Example:

Input	Output
Index 0: 1 IoP	Output 0: Freeze Address: 0.5 IoP
	Output 1: OP_Return
	Output 2: Change address 0.495 IoP

This Transaction has a fee of 0.005 IoP, which is 1% of 0.5 votes

Next Steps

- Create Java Api in new repository that will generate these transactions → Matias
- Add validation in IoP core client → Rodrigo

Client IoP Infrastructure Requirements

Status

Currently under development by Matias Furszyfer.

Next release scheduled for late November 2016 : Voting System: IoP Contributors Android App, IoP Voting Android App.

Worked at:

- https://github.com/Fermat-ORG/iop-android-wallet
- https://github.com/Fermat-ORG/fermat-framework
- https://github.com/Fermat-ORG/java-iop-profile-server
- https://github.com/Fermat-ORG/iop-ios-wallet

Working at:

- IoP Contributors Android App
- IoP Voting Android App

Introduction

Profile Server Requirements

Status

Version 1.0.0-alpha was released in March 2017.

Working at: https://github.com/Fermat-ORG/iop-profile-server

Introduction

The main characteristics of this network are:

- **Geo-localized:** For security, privacy and resilience reasons, the home network is geo-localized. Clients will choose their profile server primarily by physical proximity and secondarily by economic factors. We expect this condition to drive decentralization and have a huge number of profile servers participating in the network.

- **Incentivized:** Node operators run profile servers as a for-profit business.

Individual Profile Servers provide services to clients. That is the purpose of this network. Besides that they provide services to each other in order to act coordinated as a network. To do so they establish relationships between them. The following sections goes through the details.

Main Services Provided to Clients

Main services provided by profile servers to clients are:

- **Personal Network Profile Hosting:** The IoP is mobile first and mobile devices are not always online. Profile Servers host Personal Profiles and know when people behind those profiles are online and when they are not. In this context hosting means storing profile info and serving that info to whoever requests it. Profile hosting is a paid service. Details regarding the payment mechanisms can are explained in sections below. [to be written]

- **Application Service Calls:** Communication between end user devices is made through application services. Devices connected to a Profile Server signal which application services they are running and available to receive calls. When end users want to interact between each other using a certain app, this app will talk to a remote app through the IoP using one of the available application services running locally, which in turn must be running at the remote device. If this condition is met, the app can place a application service call through a Home Node. Application Service Calls is a paid service.

- **Profile Queries:** Profile Servers allow any entity to request profile lists and profile info in general. This is a free service below a certain threshold. Above it, it is considered for commercial use and fees are applied. Profile browsing is the natural way for end users to find other end users on the network. Specially when they need to connect for the first time. Once they are connected, both parties remember which is the Profile Server of their counterparty.

Client - Node Use Cases

In relation to a node, a client may or may not be a customer. It is a customer when its own profile is being hosted at that node. Nodes distinguish between customer clients and non-customer clients and have specific functionality for both of them.

Customer Client Use Cases

Profile servers provide certain services for profit. That means that the clients that consume and pay for those services are their customers. Nodes understand they have a customer-provider relationship and the services provided to customers are prioritized in relation to services provided to not customers. These are the use cases in this context:

- **USE CASE: Hosting Profiles:** People need their profiles to be partly hosted by a profile server because mobile devices are not always turned on or online. The whole system assumes clients will host their profiles at one of the closest profile servers from end users average location in real life. This assumption allow end users to find other end users by knowing their alias and approximate location. In the special situations where end users are usually located in more than one place at the time, clients might host the profiles in more than one Profile Server at both places. Profile servers charge for hosting end users profiles. Services related to this use case are:
    - **Be my Profile Server (Profile, Hosting Plan):** Clients can sign up for a hosting plan for a certain Personal Profile. Once they sign up, the node involved becomes the profile server to that profile. Other profiles at the same client might sign up or not with the same node.
        - [Hosting Profiles Release: In this release Hosting Plan parameter can be null since Hosting Plan functionality comes later. In this release all sign ups are accepted unless the node reached its configured limit.]
    - **Profile Check In (Profile Pub Key):** For privacy and economic reasons end users only place at profile servers the information of their profile that is needed to be online. The rest of the information they keep it at their own devices. In order for other end users to get that extended information they need to know when they are online and when they are not. The check in allows end users to set themselves online. Extended profile info can be obtained only from online clients (after connecting with them). Profiles that are online can also establish app to app connections and interact using those apps.
        - [Hosting Profiles Release]
    - **Profile Check Out (Profile Pub Key):** Clients can do a check out to announce they are going offline. Nodes automatically check out profiles when their connection is lost.
        - [Hosting Profiles Release]
    - **Update Profile (Network Profile):** Clients need to update their profile info from time to time. If the information updated is the one belonging to the Personal Shared Profile, then the profile server must update this info in its neighborhood. For that reason the frequency of such changes is limited.
        - [Hosting Profiles Release: No neighborhood functionality exists yet, so changes impact locally for now.]

    - **Move Home To (New Profile Server Profile):** Eventually a client will decide to move to another profile server. The current profile server is notified via this method call. It will remove all info of the moved profile but it will remember for a while where the profile has gone, in order to help people find the moved profile. Neighboring nodes will not be notified because it is expected that their pointers will expire before the node removes it own pointer to the new location.
        - [Hosting Profiles Release]

- **USE CASE: Connecting One Device to Another:** People need to communicate through a profile server because mobile network operators mostly disallow incoming TCP/IP connections to mobile devices. **Risks: a)** semi-malicious nodes manipulating this functionality in order to serve more users. **b)** malicious nodes executing man-in-the-middle attacks. Services related to this use case are:
    - **Application Service Check In (Application Service Type):** Clients to communicate with each other use application services. Nodes don't define application services, they just assume that there might be different application services running on client devices. When a client device gets internet connectivity, it checks them in at the profile server. This allows other clients to know which communication channels are open on that client. The application service check in occurs after the profile check in since application services are communication channels in the context of that profile.
        - [Connecting Devices Release]
    - **Application Service Check Out (Application Service Type):** When clients go offline, their profile server automatically checks out their application services. However, using this call, a client can check out a specific application service while it stills continues to be connected to the profile server.
        - [Connecting Devices Release]
    - **Application Service List (Profile Network ID):** This method returns the list of application service ids checked in by a certain profile. From this list clients learn the types of the application services they might want to call later.
        - [Connecting Devices Release]
    - **Application Service Through Node Call (Application Service Type):** A client (customer or non-customer) can ask the node to establish a connection to a customer client of the node. The node will issue a token for each client that will be used to identify the specific connection. The node will inform the callee to accept the call on the Application Services Interface.
        - [Connecting Devices Release]
- **USE CASE: Related Profiles:** People can announce relations between each other, or even relations between multiple profiles that belong to one person.
    - **Add Profile Relation (Relationship card):** Add link to another profile and prove it by a relationship card signed by the related identity.
        - [Hosted Profiles Queries Release]

- **Remove Profile Relation (Relationship card ID):** Remove link to another profile.
  - [Hosted Profiles Queries Release]

Non-Customer Client Use Cases

Nodes also serve non-customers. They do so with lower priority and only if their resources allow them to do so. If they are overwhelmed they might return *Busy* and use their resources for their customers. This prioritization has its limits, since without these services customers can not be discovered or contacted. Use cases for non-customers are:

- **USE CASE: Hosted Profiles Queries:** Profile servers need to provide information about the profiles they host. This is the way clients use to find the Network Profile information of the people they want. These are the needed services to cover this major use case:
  - **Hosted Profile Data (Profile):** Returns all the info stored for the requested profile.
    - [Hosted Profiles Queries Release]
  - **Hosted Profile Types List:** Provides a list of the types of profiles hosted at a node at the time the information is requested. The list includes the amount of profiles of each type being hosted. This is to be used for instance by network explorers web sites or apps.
    - [Hosted Profiles Queries Release]
  - **Hosted Profile Relations List (Profile):** Returns all related identities to a profile together with their relationship cards.
    - [Hosted Profiles Queries Release]
  - **Profile Search Query (filter with multiple criteria)**: Returns all profiles that match given filters. Possible filters are:
    - Is hosted on the node - true, if the result profiles have to be hosted by the node being queried; false, if the result profiles can contain profiles hosted in the queried node's neighborhood
    - Profile type - optional wildcard pattern
    - Profile name - optional wildcard pattern
    - Location - optional GPS location
    - Radius - if Location is specified, the radius within each result profile has to have its location
    - ExtraData - optional regexp pattern
    - [Hosted Profiles Queries Release]
  -
- **USE CASE: Shared Profiles Queries:** Profile servers need to provide information about the profiles of their neighbors. This is the way the Home Network facilitates clients to find what they are looking for. This use case is about people trying to find other people with a certain filter in a certain area. For example people selling a bike, people who like dancing, or similar. The filter is about location and information found at people's shared profiles on the network. Again in this case the bulk of the work is pushed towards clients. Home node network will not provide indexing services as a network to facilitate resolving these

queries at the network level. Indexing the network information might be the subject of a higher level p2p network or even a centralized service. Neighbors share part of the profile info they host with each other just to facilitate this. For example, they might know in their neighborhood all the people renting flats, but they don't have any specific information about those flats more than the price. If clients wants to learn more, they will need to visit the profile servers of those profiles for that. The network does not define which information is shared, client apps do. The network just provides a way to filter their neighborhood according to that information shared. These are the needed services to cover this major use case:

    - **Shared Profile Query:** Once clients find a node close to the target location they use this method to query the node about the information they are looking for. Note that the information on Shared Profiles might be different from one profile type to the other. That means that *name* or *alias* might not always be a field to be provided in this query. The request will put the node to search across all the Shared Profiles records it stores from all the nodes in its neighborhood including its own hosted profiles shared data. The list returned contains all matching shared profiles including the node profile of the node that hosts each of those shared profiles. To perform the shared profile query, the requestor simply performs Profile Search Query described above with "Is hosted on the node" set to false.

        - [Shared Profiles Queries Release]

- **USE CASE: Profile Changes Notifications:** These are the needed services to cover this major use case:

    - **Subscribe Me to Profile Changes (Profile Pub Key List, Relationship Cards List):** This method is used by clients that want to be notified when changes occur at certain home profiles. The node will send a notification when these profiles become online or offline, and when changes are made to the information stored by the node of those profiles. It will only notify which field changed, not the changes themselves. Relationship cards are requested to prove that clients has established relationships with the profiles it wants to monitor their changes.

        - [Notifications Release]

    - **Unsubscribe Me to Profile Changes (Profile Pub Key List):** This method is used by clients that don't want to be notified anymore when changes occur at certain home profiles.

        - [Notifications Release]

Server - Server Use Cases

- **USE CASE: Neighboring Servers:** Profile Servers knows they run in a neighborhood. They share some burden of their work with neighboring servers in order to make the individual services they provide more resilient and also facilitate clients finding the people they are searching for. By being neighbors they are also competitors, since they are competing for providing services to the same regional market. The Profile Server relies on LOC component to maintain the information about the node's neighborhood and the Profile Server receives updates from LOC when the neighborhood of the node changes. Services related to neighboring servers are:

    - **Accept Shared Profile Info List (Shared Profile List):** This method is used by neighboring servers just after they become neighbors to download list of shared profiles from their neighbors.The server being asked to share its profiles will start sending updates (using the calls

described below) of its profile database to the server that sent the request. All the information received expires after a certain time established by a system wide parameter. **Risks:** sybil attacks from nodes might disrupt a region.

        - [Neighboring Nodes Release]

    - **Add/Update Shared Profile Info (Shared Profile List):** When a new client sets up its profile with a profile server, or when a client submit an update of its shared profile info, their hosting profile server uses this method on every neighbor to further update this information.

        - [Neighboring Nodes Release]

    - **Delete Shared Profile Info (Shared Profile List):** When client's subscription to its hosting profile server is cancelled for any reason, the hosting profile server uses this method on every neighbor to delete the shared profile info at neighboring servers.

        - [Neighboring Nodes Release]

    - **Renew all Shared Profile Info:** This method is used by a profile server on each one of its neighbors in order to renew the expiration time of all of its shared profiles stored there.

        - [Neighboring Nodes Release]

- **USE CASE: Backup Servers:**

    - **Accept my Backup Partner List (Node Profile List):** This method is used to tell a neighbor which nodes are a backup in case the subject node is unavailable. The same method is used to update the backup list in case it changed.

        - [Backup Nodes Release]

    - **Backup List (Node Profile):** When a certain node is unreachable, clients will go and ask its neighbors where to find its backup. This method returns the list of nodes that will do the job in case the subject node goes missing.

        - [Backup Nodes Release]

Backup Nodes Use Cases

In this concept, we do not require profile servers to do any action. We leave the responsibility to the client. So, the client establishes a profile server somewhere and once this is done, it similarly establishes a backup node somewhere not very close to the original profile server, but not very far. Once this is done, it needs to inform its profile server and the backup node will connect with each other and keep their data in sync for that identity. It does that using the notification functionality. So, what happens if the profile server is unavailable? The identity has the list of backup nodes locally, so it does not need to search for anything and can immediately connect to one of its backup nodes. The backed up profile is searchable in the network provided that the neighbors link to a original profile server and also to the back ups. So we need a redirection here to be made. The profile server is responsible to propagate (as a part of shared network profile) profile's list of backup nodes, so that the search query not only returns the profile server, but also those backups.

- **USE CASE: Backup Nodes:** Services related to backup nodes are:
    - **Be my Backup Node (Network Profile):** This method is used by a client to request nodes to be its backup node.
        - [Backup Nodes Release]
    - **Update my Backup Node List (Backup Node List):** This method allows a client to update their list backup nodes at their profile server.
        - [Backup Nodes Release]

Cryptography

We are using public key cryptography based on Ed25519. We know that by choosing this we are not using the same that the bitcoin blockchain. We understand that public keys are not going to be able to be used as addresses to transfer tokens to, and we see this as a desired feature for privacy reasons.

Encryption

We use TLS 1.2 for encryption.

Message Protocol

Protobuf

Human readable protocols were outruled because they tend to need more bandwidth and the clients would need to spend more battery life on (de)serializing and possibly more money in mobile data plans. To debug binary protocols, some tooling is required, otherwise implementers would spend significant time trying to understand packet dumps by hand. Google protobuf is a well-established toolset for generating high-performance binary protocols. There are newer alternatives like Flatbuffer, but they do not have wide support yet in all languages. There are also older alternatives like ASN.1, but their implementations might not perform as well as protobuf does.

- Java toolset
- C# toolset: by Google and by Marc Gravell
- Rust toolset

- WireShark dissector

Principles of the Protocol

Backwards Compatibility

We need to make sure a given profile server is able to talk to multiple client versions. We expect more technical skills from the profile server operator, and they are financially motivated to support as many client versions as they can. Therefore we use semantic versioning on the protocol and we expect the profile server implementations to support clients that have the same major and minor versions plus some older versions. Clients can be expected to upgrade every year or so, but not significantly more often.

Fail-Fast

In case of a protocol breach, both the profile server and the client should answer with a protocol breach message and drop the connection - whoever notices it faster. When we have multiple implementations from different vendors on both the server and client side, we easily end up in a situation that we saw with browsers and web pages. In case of HTML, being liberal on reading and strict in writing has lead to implementations that are further and further from the protocol, therefore they needed more and more complex code that read those invalid web pages.

Message Interfaces

Profile servers talk to different user types through different message protocol interfaces. Each protocol interface exposes only a set of functionality. Home node will expose these interfaces through different TCP/IP ports. The configuration file of the node allows sysadmins to configure the port they want each interface to run. By default the unencrypted interfaces are all configured at port 16987/tcp, encrypted use port 16988/tcp. Nodes Profiles includes information about all ports configured at the node. We have identified these interfaces:

- **Non Customer Clients Interface:** It has all messages intended for non-customers. All messages going through this interface are encrypted.
- **Customer Clients Interface:** It has all messages intended for customers. All messages going through this interface are encrypted.
- **Neighbors Interface:** It has all messages to speak between neighbours. Messages going through this interface are encrypted.
- **Primary Interface:** It is the interface that other servers use as the primary contact point and where they can learn about the other ports the node is using. Messages going through this interface are not encrypted.
- **Application Service Interface:** When clients want to connect to each other, they do so via Application Services (AS). There are two modes of operation when it comes to AS. First mode is using the profile server of a callee as a bridge, second mode is directly between the peers. If the first mode is used, it is this Application Service Interface to which the clients connect and send their messages to be forwarded to the other peer. This interface is encrypted.

Messages

We avoid using the *extensions* and the *any* concepts in protobuf for now, because they are not portable among all code generators we need to use. We use functionality of Protobuf that exists in version 2 as well as in version 3. We avoid functionalities that are specific to Protobuf version.

Protocol messages can be found here:

https://github.com/Internet-of-People/message-protocol

Location Based Network Node Requirements

Status

Currently under development by Istvan. Initial test packages were already released and seeds were initialized, now documenting, trying to collect feedback, test and make the release more open to the public. .

Working at: https://github.com/Fermat-ORG/iop-location-based-network

Introduction

Node operators set a location for these nodes that becomes part of their network identity. This location can not be anywhere in the globe. Their IP must resolve to that location and its latency measurements must be consistent with it, other wise, the network will reject them. Once accepted into the network, a new node becomes part of a network of nodes which topology is based on node's geographical location.

These nodes have 3 major interfaces, each one running on different ports:

1. **IoP Servers Interface:** Through this interface IoP servers installed on the same hardware are allowed to consume node services needed to participate in this network as IoP service providers.
2. **LBN Nodes Interface:** Through this interface nodes talk to their peers.
3. **Clients Interface:** IoP Clients consume node's services through this interface.

Node to Servers Use Cases

These are services that Nodes gives to Servers running on the same hardware.

- **USE CASE: Sharing Server Interfaces:** Server need to tell nodes at which interfaces clients can contact them: The node builds up a list of its server interfaces that clients can query, see Node to Client Use Cases.
    - **Publish My Client Interface (Server Type, Ip, Port):** Provides the Ip and port of the default client Interface. This is the informations clients needs from a node in order to contact a server running on top of it.
        - [Unrelated Nodes Release]
    - **Remove My Client Interface (Server Type):** If a server closes in a controlled way or loses network connectivity, it should ask to remove its Ip and port from the list of client interfaces. When a server changes its Port or Ip, it should remove itself from the list and register again with the changed details..
    - **Get Neighbour Nodes Ordered by Distance (KeepAlive?):** Provides a list of node profiles that this node has as neighbors. Useful for sharing information with neighbouring nodes, see e.g. shared profiles of the Profile Server. If the keepalive flag is set, the connection is kept alive and changes of the neighbourhood (i.e. node was added, updated or deleted) are continuously reported to the requesting local service.

        - [Unrelated Nodes Release]

Node to Node Use Cases

The Location Based Network needs to act like a coherent entity. This means that besides these core services nodes must also provide services to other nodes.

The separation of relationships between nodes: being unrelated, colleagues, neighbors determines the accessibility nodes have between them. These relationships are always mutual for both nodes. The access level of a remote node to the functionality of a node depends on the relationship the nodes have between each other. Abuse is controlled method by method against access logs kept for each of them.

We can classify the relationship between nodes like this:

- **USE CASE: Unrelated Nodes:** Nodes join the network unrelated to other nodes. Later they can become colleagues or neighbors. Unrelated peers by default don't trust each other. Any of them can be a malicious node of any kind. That means that interactions with unrelated nodes are taken with special care. There are certain services between unrelated peers:
    - **Get Node Count ():** Get number of nodes (no matter if colleagues and neighbours) known by this node. Useful for other nodes to determine a targeted map size while initializing the world map.
        - [Unrelated Nodes Release]
    - **List Random Nodes (Count, Include Neighbors?):** Provides a list of random nodes known by the requested node. Useful to collect node contacts while a new node is initializing its own maps.
        - [Unrelated Nodes Release]
    - **List Closest Nodes Ordered By Distance(Count, Radius, GPS Location, Include Neighbours?):** Provides a list of the closest nodes known by the requested node to the location provided. Useful while exploring the world to find nodes near a location. Nodes are returned by range from the targeted GPS location in ascending order.
        - [Unrelated Nodes Release]
- **USE CASE: Colleague Nodes:** Nodes must know nodes around the globe to help clients reach faraway lands. As a network, all nodes equally benefit if they can avoid non-profitable requests from clients, such as routing requests. For this reason, nodes exchange node profile info with each other and become Colleagues. Services associated with this type of relationship are:
    - **Accept me as a colleague (Node Profile):** One node request another to exchange their profile info and become colleagues. If the request is accepted, then the requested node returns its latest profile information (to make sure no outdated information is stored at the initiator side) and both node profiles end at both node tables until the information expires. The response should also contain the detected external IP address of the initiator node node to help it autodetect its connection information to be advertised to the network.
        - [Colleague Nodes Release]

    - **Renew Colleague Relationship (Node Profile):** This method is executed by the initiator of the Colleague relationship before the expiration date. It can also be used to update the profile info (e.g. if an IP change has been detected or server was restarted and its PORT has changed). By executing successfully this method the initiator also renews the expiration date of the local information of its Colleague.
        - [Colleague Nodes Release]

- **USE CASE: Neighboring Nodes:** Nodes must know their neighborhood very well. Being neighbors is a type of relationship between nodes. Services related to neighboring nodes are:
    - **Accept me as Neighbor (Node Profile):** This method is used by nodes entering the neighborhood. When they find other nodes that qualify to be their neighbors they execute on them this method to get the relationship established. The previous relationship of the nodes may be either unrelated yet or colleagues already, i.e. there is no need for nodes to be colleagues first. The response should also contain the latest node profile information of the requested node and the detected external IP address of the initiator node.
        - [Neighboring Nodes Release]
    - **Renew Neighbor Relationship ():** This method is executed by the initiator of the Neighbour relationship before the expiration date. It can also be used to update the contact info if IP or PORT has been changed. By executing successfully this method the initiator also renews the expiration date of the local information of its Neighbor.

Node to Clients Use Cases

The network itself must provide certain core services in order for end users to find each other in the network and later communicate. These core services are defined out of these known use cases:

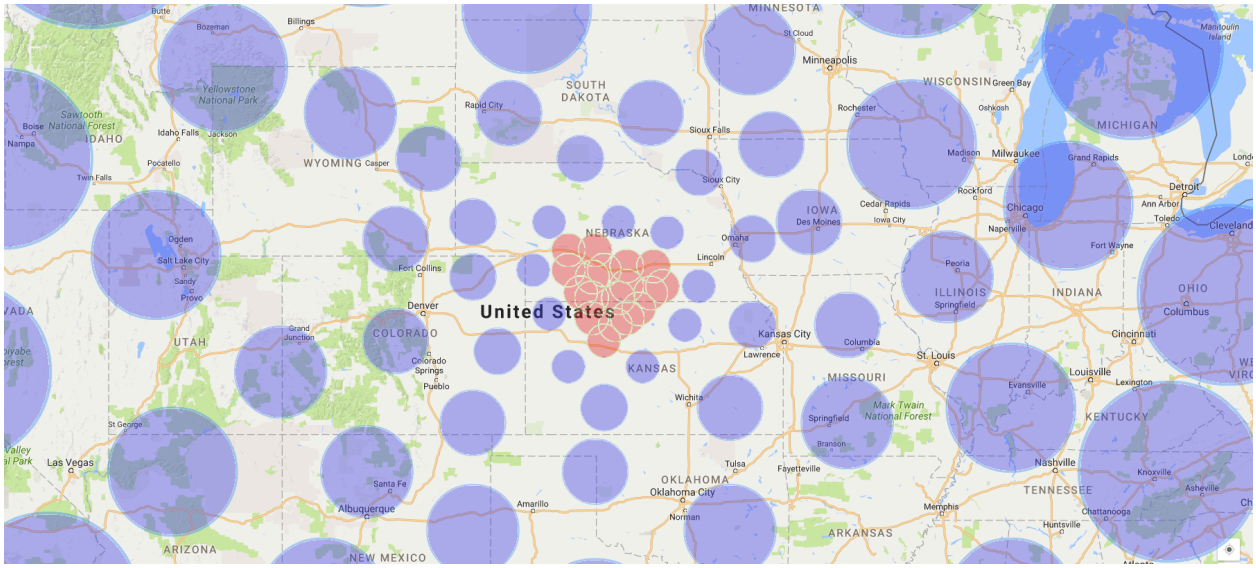
- **USE CASE: Finding a Server Public Interface:** Clients can ask anytime for information regarding the servers running on top of a network node. For those situations we have the following services:
    - **List Available Servers ():** This method returns the list of servers that previously sent their information to the network node including the IP and Port. This information was previously received when the server reported that to the node.
        - [Routing by Location Release]
- **USE CASE: Finding Known People for the First Time:** Here we assume that when people want to find someone they know, they also more or less know where they are and they can provide that information to be used by the client software to help them find them. In order for the network to scale to possibly millions of nodes, most of the work is pushed to clients, leaving the work done by nodes to the minimum possible. This means that when end users finally find each other, they must remember the server profile of the people found, because finding them again is an expensive operation for them (time wise) and for nodes (resources). This means that the use case for finding known people has a low frequency. The procedure to find known people for the first time starts at the profile server of the client performing the search and ends at the targeted people's profile server. The services to clients associated with this use case are:

- **List Closest Nodes Ordered by Distance(GPS Location, Count, Radius, Include Neighbours?):** This function returns a list of the nodes closest to a latitude / longitude pair. Using this service at different nodes, clients can navigate to a target region where the profile server of the person to be found should be. Once clients enter that person's profile server's neighborhood, all nodes will know about the target person and which of the nodes in the neighborhood is his profile server. If the searched person is not found at the expected location, the search should be continued outside the neighbourhood of the closest node, this is how setting the flag to false may be useful, There is a similar method to be consumed by nodes. The main differences with this one are the security measures implemented to avoid abuses [to be described later].
    - [Routing by Location Release]
- **Get Neighbour Nodes Ordered by Distance():** If a person was not found on a node, the search has to continue outside the neighbourhood. If closest nodes outside the neighbourhood (discovered through the previous operation) also do not know the profile, the client has to look even farther away. Doing so the client needs to know the area that has been already discovered, otherwise it might return to the same area once again. There is a similar method to be consumed by local services. The main differences with this one are the keepalive feature to receive updates and security measures implemented to avoid abuses [to be described later].

Local Network Map Algorithm

The network is location aware and every node knows only a part of the network. This local network map is unique to each LOC NODE and partly overlaps with information kept at other nodes. Each node knows better the part of the network that is closer to it and as the distance increases it knows less. Nodes implement an algorithm that helps them pick which nodes they need to know about and which nodes they don't. As the burden to announce themselves to the network is put on newcomers, already running nodes just need to wait to be contacted.

The Local Network Map consists of two parts with different goals and properties. The Neighbourhood Map aims to maintain a comprehensive list of the closest nodes. The World Map aims to provide a rough coverage of the rest of the world outside the neighbourhood. To prevent too much data and limit node density, the world map picks only a single node from each area. Covered areas are defined with circular "bubbles" drawn around single nodes that must not overlap. The World Map uses a changing bubble size to have a denser local and sparser remote node density. So farther the area, the bigger bubbles are, i.e. less nodes are included. Area bubbles are defined autonomously and specific to the LNM of each node. There is no node count limit for the World Map, the increasing size of bubbles serves as a limit for node counts. For the Neighbourhood map the bubbles may overlap, so we need an maximal count of neighbouring nodes.



Node distribution example. Distance required between nodes plotted as node bubble size for colleagues nodes. Neighboring nodes drawn in red.

To add a new node to our World Map, the new node has to agree and also include our node to its World Map. If this succeeds, they are considered to be colleague nodes. Consequently, the relation of being colleagues is mutual.

The *Local Network Map* algorithm uses a predefined formula to decide if a node should be added to the World Map. The formula described below is used against both the node that is to be included and the closest node on the LNM to this one.

Being:

A = Current node.

N = Newcomer node.

C = Closest node at the LNM to the newcomer.

Dan = Distance from A to N

Dac = Distance from A to C

Dnc = Distance from N to C

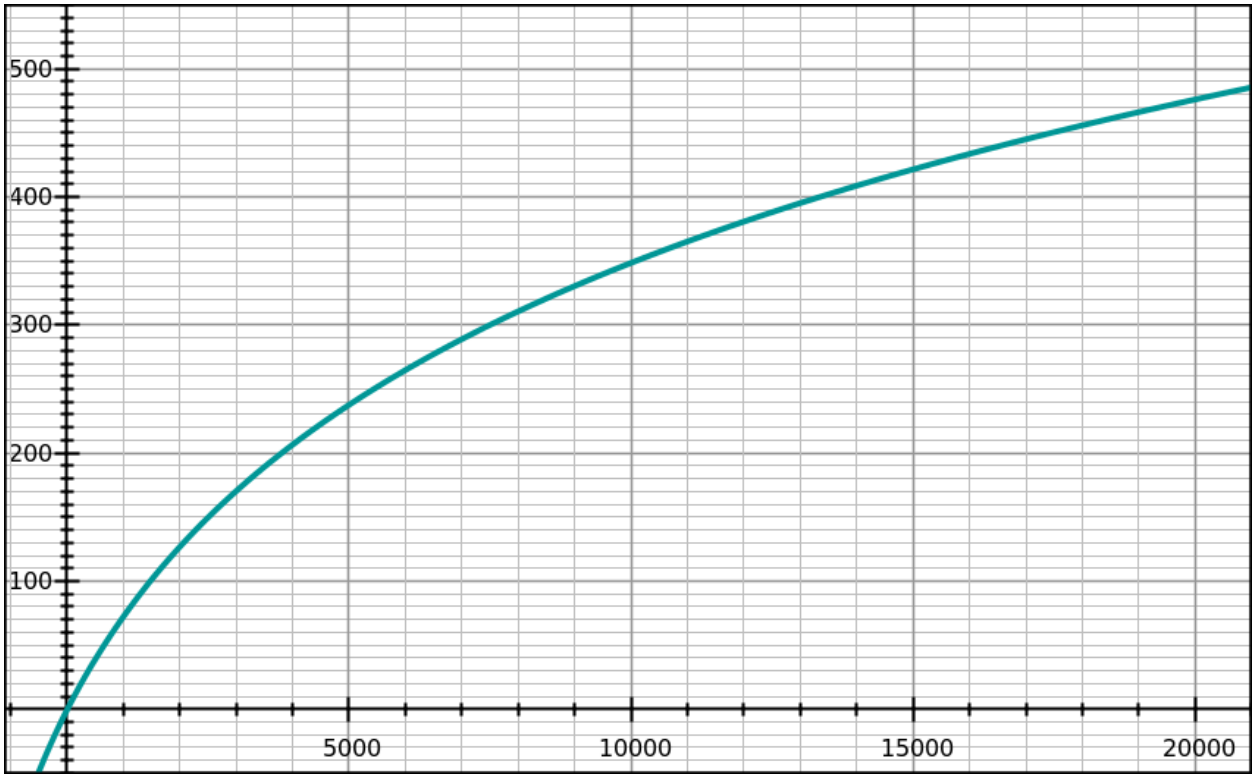
Nmax = Maximum number of nodes that will be added to the Neighbourhood Map. (estimated magnitude: 100, current exact value is 50)

d(x) = Node bubble radius at an x distance from A = log(x+2500) * 501 - 1700

Bf = Bubble radius required for nodes at (20,000 km) the farthest point on earth = (est. 480 Km)

Nbubble = Bubble radius of Dan

Cbubble = Bubble radius of Dac



In this graph the x axis is Dan [km], the y axis is the corresponding bubble size of a node at that distance.

We require that the bubble around each node must not overlap with any other bubble of colleagues nodes. To verify this requirement it's enough to check intersection with the bubble of the closest node. Thus our node inclusion criteria just verifies that the bubble of node C does not overlap with the bubble of node N. If the following formula applies then the new node is accepted.

d(Dan) + d(Dac) < Dnc

By executing this algorithm, each node would end up having a piece of a full network map, with a high density of nodes surrounding the subject profile server and the concentration decreasing smoothly as the distance from the subject increases. In this way the profile server can effectively route clients to far distances without the need to have the full network map.

For example, if:

Dan = 5000 km

Dac = 4500 km

Dnc = 350 Km

Nbubblesize = log(5000+2500) * 500 - 1700 = 237.53 km

Cbubblesize = log(4500+2500) * 500 - 1700 = 222.54 km

Nbubblesize + Cbubblesize < Dnc ⇒ 406.07 Km < 350 km ⇒ This node won't be included in the local network map because its bubble is overlapping other nodes bubble already on the local map according to the distance the new node is to the subject profile server.

Several processes are needed to keep a healthy Full Network Map distributed across different Local Network Maps:

1. **Local Network Map Initialization:** New nodes execute the Local Network Map algorithm while initializing. They use information of other nodes to scan the network and find possible candidates with whom exchange their Node Profile. They first validate using the same algorithm that a candidate is good for their local network map and after that they request the Node Profile exchange. The initialization process consist of the following steps:

    a. **World Discovery Phase:** To avoid every new node to follow the same path and go against the same set of nodes many random selections are done during this process.

        i. The first step is to randomly select a seed node from the ones available. Since seed nodes are old nodes we can assume they have their LNM without major holes and that we can ask them how many nodes they know in total. That would be our target amount of nodes.

        ii. Second, ask the selected node to provide a list of 100 randomly selected colleagues nodes from its own LNM. Though we are under the World Discovery Phase, we do not exclude neighbours because the algorithm would not work while the network is small.

        iii. Run the Local Network Map algorithm over the list of nodes and for the ones that qualify, try to exchange Node Profiles with them, i.e. try to make them colleagues.

        iv. See if the 75 % of the target was reached and added to the LNM already. If not, randomly pick one of the nodes in the current list and ask him a new list of a 100 colleagues and go back to ii) .

        v. Once the 75% of the target is added we are done. The rest will be filled either by nodes requesting to be added or by the maintenance process described below.

    b. **Neighborhood Discovery Phase:** In this phase the goal is to add to the LNM the subject node's neighborhood. The steps to do so are:

        i. Pick the closest node to the subject node on the LNM and get routed to your own location. This means acting as a client that needs to get the closest node to one location asking several nodes to find it. Continue with that closest node you have found after being routed.

        ii. Ask that node a list of the 10 closest nodes to the subject node. With this list create a first list of neighbors candidates. If the list is empty that means you are the first in that

neighborhood and no further action is required, just wait for neighbors to appear someday. If not, continue in iii).

iii. From the neighbors candidates list choose the closest node to the subject node location.

iv. Ask this node for a list of the closest nodes to the subject node. Avoid asking a node you already asked this list before. If you already asked this node before, then do it with the next closest to the subject node you haven't ask for its list yet.

v. Append the list of nodes received to the list of neighbors candidates.

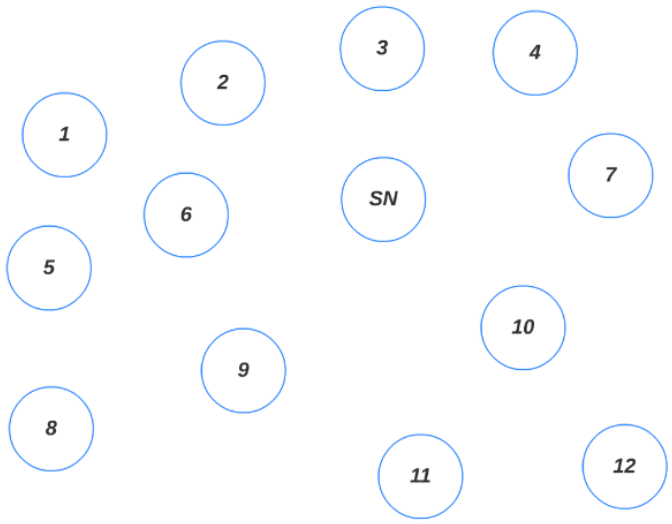
vi. If in iv) some nodes were added to the list, then goto iii) until no nodes are added anymore. We will end up with a list of all candidates nodes to be our neighbors.

vii. It can happen eventually that the number of neighbors identified exceed the Nmax value. For that reason, the list of neighbors now must be ordered by distance to the subject node (increasing). Next we need to loop from the beginning of the list exchanging profiles with these neighbors until we either exhausted the whole list or successfully exchanged profiles with Nmax nodes. Neighbour nodes must be flagged at the LNM as neighbors. It can happen that in a crowded area a new node appears and its natural neighbors already reached the Nmax threshold. In this situation they will still accept the newcomer even exceeding momentarily the threshold. They will return to the right range later, rejecting renewals from farther neighbors.

An example for Neighborhood Discovery Phase follows: (See the graph below).

1. Let's say the closest node at the subject node LNM is #1 and all the other nodes are unknown.
2. From #1 we get routed according to the Location Based Routing Procedure listed below, and in this example we end up the routing at node #2.
3. We ask #2 the list of closest nodes at our location and it returns is {3, 6} being that our first list of neighbors candidates.
4. The closest node to the subject location here is #3.
5. We ask #3 for its list of nodes closest to the subject node and it returns {6, 9, 10}
6. Now our candidates list is {3, 6, 9, 10}
7. As we added some nodes then we go to ii)
8. The closest node to the subject location continues to be #3.
9. We skip #3 and go and ask #6 for its list of nodes closest to the subject node and it returns {2, 3, 7, 10}
10. Now our candidates list is {2, 3, 6, 7, 9, 10}
11. As we added some nodes then we go to ii)
12. The closest node to the subject location here is still #3.

13. We skip #3 and #6 and ask #2 for its list of nodes closest to the subject node and it returns {3, 4, 9, 10}
14. Now our candidates list is {2, 3, 4, 6, 7, 9, 10}
15. As we added some nodes then we go to ii)
16. The closest node to the subject location is still #3.
17. We skip #3, #6 and #2 and ask #10 for its list of nodes closest to the subject node and it returns {3, 4, 6, 9}
18. Now our candidates list is {2, 3, 4, 6, 7, 9, 10}
19. No new nodes were added. It is time to move forward.
20. The process of exchanging profiles starts and we end up completing our neighbouring map.



Neighbors Discovery Phase example. SN = Subject Node

2. **Node Relationship Renewal:** Each record stored locally was generated either by the subject node requesting to be added on others node's network map or foreign nodes requesting the subject node to do so. For keeping information up-to-date, each node considers the records added by foreign nodes requests valid for a system wide constant period of time (est. 24 hours, but can be adjusted later). After that the record is considered expired and subject to be removed. Each node then is responsible for sending a renewal request to the nodes which previously accepted their request in order to extend the expiration time further into the future. It is reasonable to think that old stable nodes that arrived early enough will probably have less work than new or unstable nodes and that consequence is aligned with the resilience objectives of the network. The process to renew colleagues relationships should be run daily and follow these steps:

    i. Create a list from the LNM of all colleagues nodes which relationship will expire the following day.

    ii. Iterate through the list connecting to each node and executing the Renew Node Profile method.

3. **Local Network Map Maintenance:** Finally, nodes need to have a network map maintenance process that proactively try to fill the holes in their own map. This prevents incomplete maps when for any reason part of the map gets empty or keeps empty during the lifecycle of the network. This process is executed periodically (somewhere between the minutes and hours magnitude, currently set to 5mins) and follow these steps:

    a. Repeat the following steps a few (currently 5) times to discover potential colleagues

        i. Randomly choose a GPS location.

        ii. Pick from the LNM the closest node to that target location.

        iii. Ask this node for a list of nodes on its LNM around this target location.

        iv. From this list pick the closest node to the target location and try to create a relationship with the following rules: .

        v. If we don't have enough neighbours or this node is closer than our farthest neighbour, try to make it a neighbour.

        vi. Otherwise verify with the LNM algorithm if this node qualifies to be a colleague.If so then try to make it a colleague.

Location Based Routing Procedure

Both clients and nodes need to be routed to a certain location. Sometime this location is a city where a friend lives, other times it is the location of a node. In any situation the whole purpose of having a LNM is to allow location based routing.

The procedure always start asking a node for a list of nodes closest to a certain location. The expected amount of records returned and the maximum distance to the location provided can be specified as parameter to the method call and they very much depends on the app being routed.

Usually from the resultset received the app being routed will select the closest node to the target location and will repeat the query. It might provide the same parameters or not, again that depends on the app and the use cases that is supporting.

Some apps, after receiving the first result set with no nodes closer to the target location that the ones received before will stop there and the routing procedure is considered finished. Some other apps, will query the second closest node or even the third one just to be sure none of them have a closer node on their LNM. There is no guarantee that you will find the absolute closest to the target location.

Node operation "List Closest Nodes" always returns nodes in ascending order by distance. The client can always be sure that there is no other closer node known by the queried node instance than the first one returned.

Routing Model Properties (Routing Algorithm Test)

We would like to describe here a problem that we would like the routing model to successfully solve. Any model we suggest should be tested against this example to see if there are any problems.

Alice lives in London and she wants to find her friend Bob Doe and she only knows that he lives in Middlesbrough. The distance between them is about 340 km. Alice has her profile server in London. She doesn't know where Bob's profile server is.

The fact is (but Alice doesn't know that when Bob entered the network, he made a local search for nodes in his area and found out that the nearest profile server to his location is in Sheffield (130 km) and one more node is in Manchester (135 km), then one is in Glasgow (240 km), and one in Edinburgh (200 km). Being the cheapest offer and having a good connectivity to Manchester, he chose the node in Manchester to be his profile server.

Alice does her search for "Bob Doe in Middlesbrough", which starts at her profile server. She expects her profile server or her client device to know that Middlesbrough is a mid size city (population of 174,700), so to her query it can add something like 30 km radius to cover whole city and some area around, just to be sure Bob will be found.

Her profile server should be able to find one or more nodes that "cover" the target area. By "covering" we mean that there should be set of nodes that are somehow responsible for that area. There should not be a populated area in the world that should not be covered. Having find those nodes, Alice's profile server should be able to ask them for Bob's profile server and that information should be the part of the result of her search operation.

Alice expects to see on her screen profiles of all users whose account name is Bob Doe and who live in Middlesbrough area. She doesn't want to see profiles of people outside this area - e.g. from Manchester.

If a routing model is able to solve this problem efficiently, it can be a good model. The problem should not be solved, just by tweaking some parameters to fit this example. If it can't solve this problem efficiently, it is probably not what we are looking for as this is a very real situation and a basic search query that we want to support.

As variations on this test, we can add a node to the Middlesbrough, but either Bob will still prefer to select his profile server to be the node in Manchester, or the node in Middlesbrough will join the network after Bob established profile server agreement with Manchester's node.

Example #1: David from London searches John Doe that lives in Middlesbrough.

1. David's device request its profile server in London a list the closest nodes to Middlesbrough. 5 nodes are returned, none in Middlesbrough.

2. The closest one is 50 km from Middlesbrough, so it's density is of nodes at that range is quite high. This one return 3 nodes that are in Middlesbrough itself and some other scattered around.

3. David's device chooses the node closest to the city center and ask it yet for a final list of nodes. This one should know better since Middlesbrough is where its neighboring nodes should live. This one return 10 more nodes scattered around Middlesbrough.

4. From now David's phone starts querying these nodes, starting for the one closest to the target location. It asks for a list of John Does he knows there. Together with the list of 3 profiles that match the query, the list

of neighbour nodes covered by these results come back to David's device so the app can know the reach of its query.

5. David's app catches this list, that not only include a name and a thumbnail picture, but also the pub key of each profile, and the node profile of their profile server.

6. David's device avoid asking the second node because it was reported to be covered by the first node because they are neighbours and share profile information. So it goes to the third node which is not a neighbour of node #1 and ask the same question receiving a list of 5 profiles that match the query and another list of covered neighbour nodes.

7. The app running in David's phone believes this is enough to show a screen with results, so it stops there and displays the list on David's phone.

8. David see the list, scroll down forcing the app to search for more. In this situation the app continue asking node #4, #5 and showing the information on the screen.

9. At some point David recognizes the picture of his friend and clicks on it.

10. The app then goes to its node cache and gets the node profile of the profile server of the profile selected.

11. It connects to that node and request the profile data of that profile. The node returns a bunch of information it stores for that person, including more pictures, texts, etc. It also returns a flag that means that John Doe is online.

12. The app opens a detailed profile screen that shows that content and presents David the option to connect with his friend. Once David presses the connect button, the app requests application service call so as to connect with the same app on John's phone. The node receives that request and arranges the call. Next steps: out of scope.

Content Address Network Requirements

Status

Working at:

- https://github.com/Fermat-ORG/iop-content-address-network
- https://github.com/DeCentral-Budapest/go-ipfs

Introduction

Strictly speaking the content addressed network (CAN) only supports 2 operations:

1. **Upload:** Client stores some content on the network and gets back a hash only based on that content. Note, that if different nodes store the same content at different times independently from each other, they get back the same hash, because that hash is only dependent on the content itself.

2. **Download:** Clients presents a hash for the network and therefore can retrieve the content that generates that hash.

This simple interface has many advantages:

- Uploading content will form a so-called swarm for its hash. If someone else tries to upload the same content, no matter how they got the content, it will join the existing swarm. This means the network de-duplicates popular contents, the download requests will be shared among all machines that have that content. The content itself will move through the network only when someone actually downloads it.

- The downloading nodes are able to cache popular contents without worrying about invalidation, because the content is immutable.

- Anyone who has an interest keeping some content available for the network is able to do so without a permission from any parties.

- It is enough to sign or just share the hash of a larger content on an external system, because it is equivalent to signing or sharing the whole content.

This simplicity has some drawbacks, too:

- Cannot share a hash in an external system and then update the content without re-sharing the new hash of the new updated content.

- Cannot delete a compromising uploaded content from the network if others still have an interest keeping it on the network

For fixing the first drawback, we need a key-value store a.k.a naming system. The second drawback is by design, we need to address that in the application layer or in real life.

The Naming System

We need a way of creating a mutable link into the CAN, so that the clients could lookup the current version of a mutable content. We also need to authorize changes to that link. In a decentralized system, signing an operation is the usual form of authorization. Looking at 2 different versions of that link, we also need to decide which version is more recent. All of these are solved by a key-value store, where the key is deterministically calculated from the public key of the key-pair used for authorizing changes, and this is used as a name for a mutable content, and the value consists of the following fields:

- hash of the current version of the content on the CAN
- sequence number that is increased with each version
- signature on the 2 previous fields made using the private key that belongs to this name

So for each mutable content we either need a separate key-pair or we would need a way to know which name can be changed by which keys. The former design will be implemented in our CAN. The later is also a solved, but slightly more complex design, which needs an immutable log of security related events, usually implemented as a blockchain.

The Proposed Implementation

Re-implementing these systems from scratch is a considerable effort. It seems to be more efficient to build on top of an existing system like IPFS (see https://ipfs.io) and adapt it to our needs. The IPFS has 2 independent, but compatible implementations in JavaScript and Go. The Go implementation is more mature currently and provides us with a CAN without significant changes. For our naming system use-cases it has a solution called IPNS that is not suitable for our needs, so we need to spend some efforts extending it.

Use-Cases for Authenticated Services

Establishing a relation between a CAN node and a service happens out-of-band, it does not happen through the network. Every service must either have a CAN node on the same system, or have an agreement with a CAN node. Therefore there is no need for a special "register service public key" message, the public key of the profile service needs to be just added to the configuration of the CAN node. All other messages on this interface needs to be authenticated with some of those configured service public keys.

- **Publish Content:** Input is an opaque byte array. If the content was successfully published on IPFS, the client gets back the hash IPFS calculated for that content and the CAN makes sure this content is available on the network at least as long as an unpublish message is received for this content.
- **Unpublish Content:** Input is the hash returned by the "publish content" message. Calling this message frees up storage and bandwidth for contents not needed any more for providing the service.
- **Publish Name:** Since the profile service knows which application profiles are paid for and are valid, it must register the public keys of those profiles on its CAN node. This is done when the first profile publishing is done. Input is a record serialized into a well-defined format that contains the signature. Output is either just success, or in case there is a more recent version already on the network, the sequence number of that version is returned in an error. Other possible errors might contain problems with the signature or the hash does not point to a valid application profile version.

- **Unpublish Name:** Since the profile service knows which application profiles are cancelled, it must stop publishing the name as soon as the application profile leaves the profile service. In case the profile was moved to a new profile service, unpublishing the name on the old CAN node saves bandwidth for the whole network. Publishing a new content for the same name on the new CAN node will succeed even without unpublishing it on the old node, but it wastes resources.

Use-Cases for Anyone

- **Provide Node Addresses:** Input is empty, output is a random selection of 5 CAN nodes this CAN node is connected to.
- **Download Content:** Input is a hash of a content. Output is a byte array.
- **Download Name:** Input is a name. Output is a record serialized into a well-defined format that contains the hash of the current version of the content, the sequence number of the current version and the signature authenticating this version of the name.

Typical Scenario

In the typical scenario for creating a new application profile on a profile service,

- The CAN node configuration contains the public key for the profile service or others IoP services in general.
- The profile service authenticates itself on a network connection to the CAN node.
- The profile service calls "publish content" with a serialized format of the application profile.
- The profile service asks the client to authenticate this change by sending the hash of the new profile data to it.
- The client might optionally download the content and check whether it agrees with the changes.
- The client sends the profile service a signed message that contains the hash of the profile data and a sequence number that is incremented with each profile update done by the client.
- The profile service sends this signed structure to the CAN node in a "publish name" message.

Proximity Network Requirements

Introduction

The primary purpose of the proximity network is ability to locate and contact its *active users* using the knowledge of their approximate current location. *Active users* are the users of proximity network that submit their activity to a proximity server. Proximity servers form neighborhoods (just like profile servers do) and distribute informations about their users activities. This allows the activities to be found by asking any proximity server within the neighborhood of the proximity server that receives the activity information from the user. We call such a proximity server the *primary proximity server* of the activity.

Activities in the proximity network have very short lifetime. When an activity is created, its expiration time is set and it can not be set to more than 24 hours in the future. If an activity takes longer than 24 hours, its expiration time has to be actively updated by its creator.

Many activities are expected to be updated very often. In these cases, we expect their creators to be connected to the primary proximity server throughout the whole period of time during which the activity is going to be updated.

The current location of an activity can change in time and this means that its primary proximity server can change as well. The activity's primary proximity server is always the one that is geographically closest to the activity's proclaimed location. Every time an action location update request is sent to its primary proximity server, the proximity server checks whether it is the closest proximity server to the new location. If it is not closest and the difference is significant, it will refuse the update and will redirect the user to the closer proximity server, which will become the activity's new primary proximity server. The former primary proximity server also deletes the activity immediately.

The functionality of a proximity server can thus be summarized as follows:

- Join a proximity network neighborhood, obtain activities from neighbors and receive updates of those activities from neighbors.
- Accept new activities from clients for which the server is the closest proximity server.
- Allow updating of the activities by their creators provided that the primary proximity server should not change.
- Allow deleting of the activities by their creators and delete expired activities.
- Propagate new activities and their updates to neighbors.
- Allow all clients to search for activities within the server's neighborhood.

Network Design

Similarly to how profile server network is formed, the proximity network also relies on Location Based Network to provide information about neighborhoods. Proximity servers thus only care about the synchronization of their databases and do not care about how the neighborhoods are formed.

Activity Description

Each activity is described by following properties on its primary proximity server:

- Version (binary, 3 bytes) - Semantic versioning of the structure, currently set to 1,0,0.

- Activity ID (integer, 4 bytes) - A unique identifier of the user's activity. The globally unique identification of an activity is this activity ID combined with the user's network identifier. It is user's responsibility not to present two activities with the same activity ID to the proximity network.
- User's network identifier (binary, 32 bytes) - This is the same network identifier as the user uses in the profile server network - i.e. it is SHA256 hash of its public key.
- User's public key (binary, 64 bytes) - Public key that identifies the owner of the activity.
- Contact information to the user's profile server - Network ID (binary, 32 bytes), IP (binary, up to 16 bytes), primary port (binary, 2 bytes).
- Activity type (UTF-8 string, up to 64 bytes) - It is expected that compatible application will create same types of activities and it thus will be easier for them to find matching users. The Internet of People Consortium dealing with standards, keeps a list of standard activities. Apps might use these standards or not.
- Current geographic location and its expected precision (2x 32-bit signed integer for latitude/longitude, 32-bit unsigned integer for precision specification) - In case of update of these values, the changes will NOT cause a propagation of the update to the proximity server neighbors. The precision information is in metres and must be between 0 and 1,000.
- Start time (64-bit signed integer) - Time when the activity starts. This can be in the past for already running or past activities as well as in the future for future activities.
- Expiration time (64-bit signed integer) - Time when the activity expires. This can be no further in the future than 24 hours from the last update of the activity. This is another field that is not immediately propagated to neighbors if updated.
- Extra data (UTF-8 string, up to 2048 bytes) - This is a user/application defined field with similar purpose to extra data field that is used in the profile server's profile. It is expected to contain a semicolon separated list of key-value pairs. Some of the keys are expected to be standardized. There are many things that this field can contain depending on the type of activity. To name some: search tags that will allow other users to find the activity easily; link to CAN network for rich content description of the activity (including formating, photos etc.); information about the route in case the activity is represents a certain checkpoint of a longer route - this may contain activity IDs and/or contact information to primary proximity servers of other activities of the same event; speed; movement direction; altitude; … The Internet of People Consortium provides a set of standard fields to be used. Different apps adhering to these standards can interoperate.

The whole activity description is propagated to proximity server's neighbors. The following are the differences between what the primary proximity server stores about the activity and what the neighbor proximity server stores about it:

- Additionally for each activity, the neighbor server stores contact information to the primary proximity server - Network ID (binary, 32 bytes), IP (binary, up to 16 bytes), primary port (binary, 2 bytes).
- Geographic location information is not the activity's current information, but rather its last known location from the last update the neighbor received. This is because the rapid changes in the location are not propagated to neighbors.
- Expiration time is the time when the activity can be deleted from the neighbor proximity server and the primary proximity server is responsible to update the activity if its expiration time has been updated before it expires on the neighbor server.

Activity Primary Proximity Server Migration

Proximity servers are happy to act as primary proximity servers for activities that are close to them - more specifically, the servers require that there is no other proximity server closer to the claimed location of the activity. If this is the case, the activity can be created on the proximity server and it becomes its primary proximity server.

When the activity location changes and its owner sends an update to its primary proximity server, the server still wants the activity to satisfy the condition of its primary proximity server being the closest to its location. Therefore the server may refuse the update if it finds out that another proximity server is closer to its new location. The client is responsible for creating the activity on the closer proximity server (client uses LOC nodes to obtain information about proximity servers in certain area). However, to prevent the activity to being migrated to another server often when it is just in the middle of two proximity servers, the primary proximity server will not refuse the updates of the activity location up to a certain threshold, even if there is another proximity server that is slightly closer to the activity's new location than its currently primary one. This threshold here is set to 10 %, which means that the primary proximity server will refuse the activity update if its distance to its new location (ignoring the precision information) is greater than 1.1 times its distance to its closest proximity server.

It is the responsibility of the activity creator to migrate the activity to a new proximity server before its current primary proximity server refuses the update. This is because the refusal of the update invalidates the activity and it is immediately deleted from the network and if the client waited for this to happen and only then created the activity on another proximity server, there would be a time gap during which this activity would not be available on any proximity server. The client should be aware of the proximity servers in its activity's area and if it finds the activity getting closer to the 10% threshold, it should migrate the activity to the new proximity server and delete it from the old one.

To prevent problems when a certain proximity server misbehaves and the client would thus have a problem migrating, the client is allowed to specify a list of servers that the proximity server should ignore when checking whether the activity should be migrated or not. The client is expected to try to migrate to the new (closer) server first and if this fails, it has an option to send an update and not being forced to migrate to the bad server.

Client - Server Use Cases

- **USE CASE: Activity Management:** Proximity Servers allow active users of the proximity network to submit new activities to the network and edit or delete them later.
    - **Create Activity (Activity Description, List of proximity servers to ignore):** Creates a new activity on the proximity server. The server checks whether it is the closest proximity server for the new activity's location and if so, it becomes the activity's primary proximity server. The server propagates the activity to its neighbors. The proximity server will not refuse the update if it is not the closest proximity server to the activity's location but at the same time all closer servers are on the list of proximity servers to ignore.
    - **Update Activity (Activity ID, Changes in activity description, List of proximity servers to ignore):** Modifies the activity. Based on which parts of the activity are modified, the server may or may not propagate the changes to its neighbors. The proximity server may refuse the update if the new location of the activity suggests that it should be submitted to a new primary proximity server. In such a case, the server provides the contact information of the other proximity server (that is closer to the new activity location) and removes the activity from its database and propagates this information to its neighborhood. The proximity server will not refuse the update if the suggested new primary proximity server is on the list of proximity servers to ignore.

    - **Delete Activity (Activity ID):** Deletes the activity from the proximity server, which also propagates the information to its neighborhood.

- **USE CASE: Activity Queries:** Proximity Servers allow any entity to search for activities registered with the queried proximity server or its neighbors.

    - **Activity Query (filter with multiple criteria):** Returns all activities that the proximity server is aware of that match given filters. Possible filters are:
        - Activities for which the server is primary proximity server - true, if the primary proximity server of the result activities has to be the server being queried; false, if the results can contain activities from the whole neighborhood of the proximity server being queried
        - Activity ID - optional ID of an activity if the client is only interested to get information about an activity which ID is known to them - it must be used together with User network ID
        - User network ID - optional network identifier of the network user that created the activity
        - Activity type - optional wildcard pattern
        - Start time not after - optional requirement for the activity not to start later than a specific value
        - Expiration time not before - optional requirement for the activity not to end later than a specific value
        - Location - optional GPS location
        - Radius - if Location is specified, the radius within each result activity has to have its location; note that due to use of precision information, the exact location of the activity is not precisely known and therefore the activity is assumed to be anywhere within its precision radius; the activity does match the search query if the query's location and radius has non-empty intersection, which is when the distance between the query's Location and the activity Location is less than or equal to the sum of query Radius and location Precision
        - ExtraData - optional regexp pattern

Server - Server Use Cases

- **USE CASE: Neighboring Servers:** Proximity servers neighborhoods are very similar to profile server's neighborhoods. See description of profile servers neighborhoods for more details.

Reputation Network Requirements

Main Services Provided to Clients

In order to provide their core services, reputation nodes need to store reputation information. This information is provided by different parties and accepted or rejected by reputation nodes based on the IoP protocol. Services provided by reputation nodes are:

- **Adding Reputation Records:** The network assigns a custodian node for the reputation of each entity. This custodian node is backed up by other nodes for redundancy. Submitters can ask any node to point them to the custodian node for a certain entity. Custodian nodes can be used to submit a reputation record as long as the record submitted complies with the protocol rules. Submitting reputation records is not for free. The fee charged to the submitter is proportional to the time this information should be stored and size of the review itself. The network self-organizes to define who is the custodian of what.

- **Reputation Queries:** Custodian nodes can be queried to get all the reputation records of a certain entity they are responsible for. Queries are for free unless the amount of them exceeds a threshold as they will be considered for commercial use.

- **Web of Trust:** Home nodes allow people to flag other people endorsing them. This endorsement is interpreted by apps in different ways. The intention is to allow people to identify which are the real profiles of other people and not clones or fake profiles. This endorsement is voluntary and at a conscious level done by people which can prove they have an established relationship with the ones being endorsed. This is proven with a relationship card signed by the party being endorsed after a relationship is established. The endorsement can be at the profile level, or at some of its items. People that endorse others usually do so after they successfully communicated with them and they are sure the person they talked to is the real life person they know.

The network does not only handle people reputation but also it stores ratings of products, services and companies. The functionality required for each entity is:

- **People:** To add a record to some people history submitters must prove they have a relationship with them. When people connect to each other via the IoP they exchange a signed relationship card. This card contains the public keys of both parties, the type of relationship and the signature of the person that later can be rated by his counterparty using this card. The submitted reputation record is valid only if it is signed by that counterparty.

- **Product or Service:** To rate or review products or services, submitters must prove they bought those products or consumed those services. In order to do that, they need a signed card from the seller where both the buyer and seller public keys are present as well as the product or service id.

- **Company:** The same card given to buyers of products or services can be used to rate companies.

The reputation p2p network is economically self sustainable since its operation is covered by the fees nodes charge. The resources they need -like storage- are part of their business model. Submitters pay in advance for the network to store their information for a finite amount of time. That means that the network will maintain itself purging expired records making the overall operation scalable and sustainable over time.

Client - Node Use Cases

- **USE CASE: Web of Trust:** As online identity can be easily faked, profile servers provide the means to allow endorsements of any of the profile it hosts, or some of its items. For example, a “Musician” profile could be endorsed as a whole or at the item which states that this person is a “Guitar Player”. Relationships between Personal Profiles are by default private and stored at client devices. Only the parties involved know about these relationships. But if any of the parties endorses the other, this is at the expense of making the relationship public. The same happens when one of the parties rates or reviews the other on the Reputation P2P Network. To avoid an unexpected privacy breach, relationship cards have a flags that signals if the card can be used for endorsement, rates or reviews or not.
    - **Endorse Profile (Profile):** Allows anyone holding a relationship proof to endorse a profile. After people accept other’s connection request, they establish a relationship. That relationship might evolve and go through different levels over time, but at each step, they exchange a signed relationship card, which constitutes a valid proof to endorse, rate or review the other party. Home nodes don’t know about relationship types or levels. If there is a valid card they allow the endorsement and saves the card as a proof that anyone can later verify.
    - **Endorse Profile Item (Item List):** The only difference here is that the endorsement is attached only to the items listed.
    - **Profile Endorsement List (Profile):** Lists all endorsements a profile has, returning enough information to locate the profiles of the endorser and to validate their signature.
    - **Profile Item Endorsement List (Profile, Item):** List all the endorsements a profile item has.

Minter Network Requirements

The Minter Network is a p2p network of minter nodes and some of these nodes are the recipient of newly issued IoP tokens. To mint IoP tokens it is not enough to run the IoP Blockchain Node. In fact the IoP Blockchain Node processes transactions and its revenue stream is tied to transaction fees only. To mint IoP tokens node operators need to run all IoP nodes at the same time and on top of that be invested in IoP tokens.

The IoP Blockchain Network is not technically mining, but processing transactions. The equivalent to what PoW does in bitcoin is the work done by these minter nodes. Conceptually in every cryptocurrency system a minter is the one who receives the rewards of the issuing of a coin in exchange for some valuable work done on the system. In IoP we state that instead of PoW like in bitcoin, IoP minters do auditing work on other nodes in the network. This auditing work has two purposes, the first one is to determine who is eligible to mint IoPs, and the second one is to ensure the quality of service the IoP network is providing to customers.

At runtime nodes in this network have a rank given by their peers. After that auditing work is done by a node, other nodes will rate it and consider the possibility to upgrade its rank. Once nodes reach the highest rank, they are entitled to get newly issued IoP tokens in a lottery.

In short, on IoP we replace PoW for PoA (Proof of Audit) + PoS. The PoA is the fair share of auditing work all minter nodes do on other nodes in the network.

Nodes Ranks

Running the Minter Node software and all the other IoP nodes it is not enough to enter the Minter Network. New entrants are ignored by their peers unless they have some stake on the system. This means a wallet balance with at least 1 IoP token. Once this condition is satisfied, the network will welcome them at the rank of New Full Node.

To prevent and discourage attacks, the network requires operators a combination of both time and money to be invested before becoming a minter. A minter is the highest rank that can be achieved and is the only one that participates in the lottery for newly minted IoP tokens.

This arrangement allow us to reward whoever is contributing most to our system, promotes decentralization and incentivizes investing in the tokens themselves. At the same time it allows us to control the quality of service provided to end users since the rules to check that the nodes are really running can also act as automated quality control and enforce a certain service level.

Nodes Lifecycle

Once nodes are welcomed into the network, the lifecycle they go through is the following:

- **New Full Node:** Every node entering the network is considered a New Full Node by its peers. It hasn't proven anything yet to the rest. To enter the network node operators have just to download all IoP node software and run them. To be accepted by the Minter Network the new entrant must be running from an IP subnet (to be defined) not used already by any other node already in the network. Everyday a set of tests are going to be run against the New Full Node by other nodes in the network to see if they can certify the New Full Node qualifies to be upgraded to the next rank level. These test includes:
    - **Blockchain Node:** Set of tests to see if the blockchain node is operating normally.

    - **Profile Server:** Set of tests to see if the Profile Server is operating normally and have already enough customers.
    - **Proximity Node:** Set of tests to see if the Proximity Node is operating normally and have enough activity.
    - **Reputation Node:** Set of tests to see if the Reputation Node is operating normally.
    - **Proof of Stake:** Verify that the node has at least 1 IoP token in its wallet.

- **Full Node Candidate:** The Full Network will upgrade a New Full Node to a Full Node Candidate after it passes all the defined tests for each of its IoP nodes for 30 consecutive days with the accepted level of failures. The day counter is reset every time the audits on a node fail. To be upgraded the stake required is of 10 IoP tokens.

- **Real Full Node:** The network considers a Candidate to become a Real Full Node after 30 more consecutive days of passing all the tests without failures and a stake of 100 IoP tokens.

- **Minter Node:** After 30 more days of successful tests a Real Full Node can be upgraded to be a Minter Node, the highest rank a node can get. The stake needed before the upgrade is of 1,000 IoP tokens. Only 50% of the most invested Minter Nodes are able to participate on the lottery and effectively mint IoP tokens. The winner's address is the one used at the coinbase transaction of the next block processed by the Blockchain Network. If audit tests fails while a node enjoys the Minter status it can be downgraded by its peers to Real Full Node rank and forced to wait 30 more days to be upgraded again.

- **Banned Node:** Some tests run against nodes can identify scammers. When this type of test is positive, the Minter Node tested is banned by their peers for an incremental period of time, starting at 1 day, and if it persists, upgraded to 1 week, 1 month, 1 year, 1 decade or 1 century. Banned Full Nodes are later ignored.

Proof of Audit

The work of Minter Nodes in IoP is about auditing other nodes in the system. We don't want or need every node to audit or be audited by every other node in the network. That would not scale. So we need to find a way that minter nodes only audit a certain number of other nodes and that should be enough.

The IoP system has several independent p2p networks, each one with potentially a different amount of nodes. If we want the work minters do to be beneficial for the entire system, all nodes on all networks should be audited, and not only nodes belonging to a potential Full Node setup. At the same time, only audits done over other minters contribute to the selection process of who deserves to mint IoP tokens. From here we might conclude that there is some mandatory work that needs to be done, probably on a deterministic way, and some other that could be more random.

Auditing Minters

There are several goals we want to achieve:

- **Scalability:** We don't want all minters to audit all others, this would not scale. So each minter should be audited by a subset of the network only. For obvious reasons this subset shouldn't be chosen by itself. The way to choose the subset must be deterministic, in the sense that any node running some algorithm should

be able to have no doubts regarding which nodes needs to audit each other node. Finally this relationship between nodes must no be the same over time, to prevent corruption at the node operator level.

- **Selfish Audits:** We want to prevent minters falsely stating others are not doing their job. As a strong reward is involved (token issuing) there is a strong incentive to cheat to get that reward. One way of cheating is preventing the rest to qualify as minters.

Auditing Non Minters

While auditing non minter nodes (nodes of any network not running under a full node configuration) there is no incentive to do it wrong, but if there is also no incentive to do it, then many lazy minters would skip this job.

Auditing Cycle

We establish a daily auditing cycle. This means that every node in the mining network is going to be audited once a day. Being audited means that all the nodes that are defined to audit a certain node, must run their auditing tests during that day. To keep it simple, we will request that each node should be audited by 24 other nodes and we will try to design an algorithm for this to occur at a rate of 1 per hour approximately, so as all of them can finish their tests within these 24 hours and at the same time the audited node is not stressed and it should be responsive during all day. If all audits are passed then the audited node can be considered clear for that day. Several situations might occur under this scheme:

- **Missing Auditors:** In an optimistic scenario all nodes do their job and do the audits. It can also happen that one or more of these nodes for any reason, don't do their job. Besides punishing the missing auditors there should be a way for the audited node to not be affected by this situation. Re-arranging the auditing set on the fly might open the door to some possible attacks. It is better to assume that no more than 4 auditors would go missing on a certain day and that 20 auditors is enough to clear the day. In the event there are not even 20 the network would not be able to count that day as a cleared by the audited node, but neither treat it as an audit failure (unless the audits that were done did fail).

- **Audit Failures:** An audit fail is the result of one node running its audit tests toward another node and these tests are not successful, meaning the score doesn't reach the pre-defined success level. In order to prevent selfish audits, a failure must be verified by the other nodes in the auditing set. This forces other nodes in the set to do the extra job of running their tests immediately after the failure was detected, in order to share the same state of the audited node. The first one to verify the failure is the next node in the schedule (or the first in the set if it was the last in the schedule). After it finishes the next one continues, and then the next one again. At that point 3 independent nodes have run their tests. The audit will pass if 2 of 3 say it passed. All auditing results and the way is was obtained is recorded on the Minters Network, so after time if there is a node misbehaving with its test results in a consistent basis it will be clear which one it is and can be banned.

- **Audit Success:** An audit is considered successful if the tests run at all nodes of a full node configuration add to a final score of 95/100. Audit successes are not immediately verified by other nodes. When 3 nodes have to audit another because the first one declared an audit failure, if two of them declare success this counts as a success for that audit.

- **Day Cleared:** A Minter Node under a full node configuration is considered to have cleared its day if all of its auditing nodes present for that day reports an Audit Success. If some auditors were missing, it needs at least 20 for the day to be considered cleared.

- **Day Not Cleared:** A day that is not cleared impacts of the node's rank, according to the Rank Promotion & Degradation Rules listed below.

- **Untested Day:** When the minimum quorum of 20 auditors fails to be reached, an Untested Day is declared. These days are skipped on the Rank Promotion & Degradation Rules because it was not the audited node fault that quorum was not formed.

Rank Promotion & Degradation Rules

Every Minter Node keeps a local copy of the history of the Minter Network. From that history is tracked not only the audits that everyone has done to each other, but also the ranks nodes have at any moment in time. There are two processes that nodes execute once every day to see who needs to be promoted and who needs to be degraded.

To make this procedures more resilient to improvised attacks, we define that promotions have to be agreed by higher ranking nodes from the ones to be promoted, and degradation has to be agreed by lower ranking nodes that the ones to be degraded. This would mean that New Full Nodes can not be degraded because they are the lowest possible ranking, while Mining Nodes can not be upgraded because they are the highest possible ranking.

Let's take a look to these procedures in detail:

- **Promotion Process:** Once a day this process is executed by all nodes that can promote other nodes (those who are not New Full Nodes). The process calculates and validates all promotions for the day according to the local history of the network recorded at every node. Later this information will be shared with their peers to reach a consensus that most of the network agrees on the same results.

- **Degradation Process:** In a similar way once a day every node that can participate (here Minters are excluded) calculate and validate who should be degraded. Later this information will be shared to try to reach a consensus on the nodes to be degraded.

Auditing Set Formation

The set of nodes to be audited is deterministic. That means that a node can not choose who to audit.

Randomization of Minting Reward

In order to prevent that all the issuing of new tokens goes to the highest stake holders only, it is necessary to introduce some randomization, allowing the tokens to be issued also to other nodes even if they are not the biggest stake holders. In this regard, we introduce a rule for the coinbase transaction:

1. 50% of the tokens are for the minter node that wins the lottery.
2. The rest is divided into up to 50 nodes this minter node audited successfully.
3. The 50 nodes are the ones with higher stake.
4. The 50 nodes must have at least Real Full Node rank.