GLOBAL CONNECT PROJECT

<ARCHITECTURE OVERVIEW>
<PERSISTENCE AND CONTROL LAYERS>

WWW.PARADIGMA.CL
INFO@PARADIGMA.CL

AUGUST 31, 2018

Contenido

ABSTRACT

The blockchain model in the maritime transport sector allows the creation of decentralized platforms on which to trace the history of a product, the process of the supply chain, enhancing trust and collaboration among all stakeholders and without existing a central entity that controls the process. All those involved can jointly create a fingerprint that is updated every time it interacts in some way with an element on its way to the final consumer.

Involves all agents in the supply chain, and provides tracking information capacity, works in conjunction with IoT devices such as sensors and uses smart contracts to model agreements between different parties, ensure that such agreements are satisfied and controls the compliance to the regulations.

This described model has the capacity to improve efficiency and reduce costs in different areas of operation. The use of an immutable and reliable registry as a blockchain makes it possible to guarantee who is responsible at all times for the transported goods.

Several companies normally participate in an international shipment of goods since several means of transport are used. All of them have their independent databases where they update the status of the shipment based on the information provided by the others or their agents. Blockchain has a clear applicability here that would make the system simpler, more transparent and less expensive. To begin with, the database (the blockchain itself) would be shared by all the intermediaries and by the sender and recipient, reducing costs. Confidence among them in general would not be necessary for this. When the container in question arrives at a certain port, an update to the database will be added. Since all the updates are signed with the private keys of the delivery and the one collected and with the key of the container, this update would act as a cryptographic test that the container is now in the possession of the port administrator. In addition, the system includes timestamps to track.

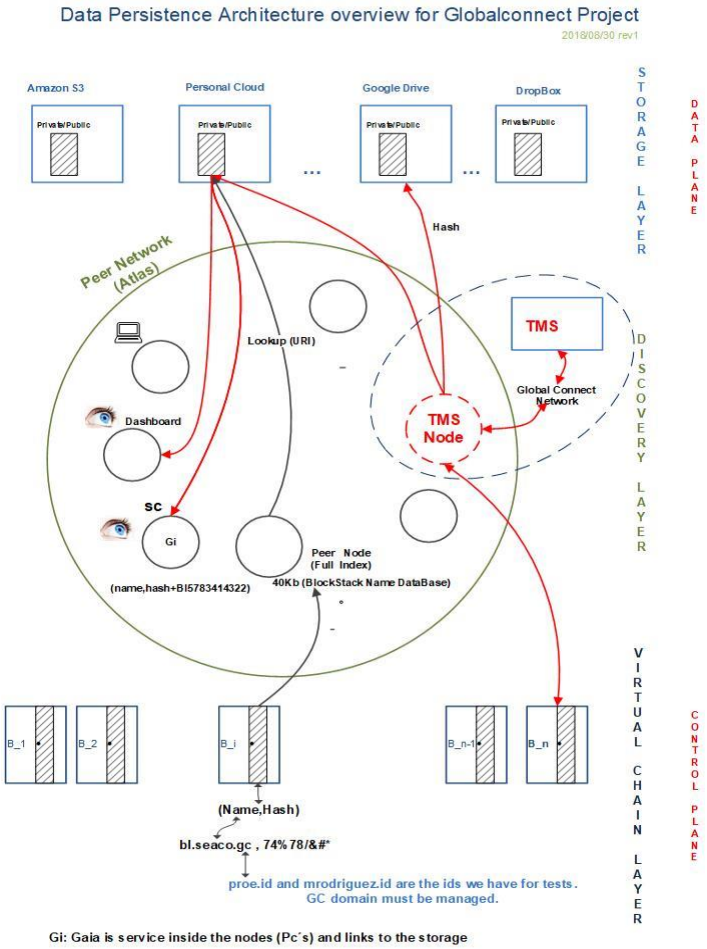
The transparency and reliability of the concept would undoubtedly help in the resolution of disputes among the participants. If an IoT device is added to this innovative approach, even more efficiency can be gained. Containers and exchange sites can have built-in devices that fully automate the process, reducing costs even more and reducing the chances of error and fraud.

ARCHITECTURE OVERVIEW

For the proposed solution we have opted by a closed ecosystem, a private and permission networks. The members are invited to join and keep a copy of the ledger Blockchain technology. The access to the respective network is be restricted and security is thus heightened.

The architecture is based on the Blockstack framework that implements the decentralized authorization through a DNS and PKI, can be used to share IoT device node addresses and data without a central authority in a trustless manner, and distributed storage in the cloud, which allows us to have scalability.

Blockstack domains (called BNS) are not registered on the traditional DNS run by an organized called ICANN. Instead they're registered on a blockchain in a fully decentralized way. This means that Blockstack domains are truly owned by their owners and cannot be taken away. All Blockstack domains have public keys by default (public keys are required to own the domains).



Architecture Overview

Our implementation has three components:

a) A *blockchain*, implemented using *virtualchains*, is used to bind digital property, like domain names, to public keys. Blockstack's blockchain solves the problem of bootstrapping trust in a decentralized way i.e., a new node on the network can independently verify all data bindings.

b) A peer network, called *Atlas*, gives a global index for discovery information and

c) A decentralized storage system, called *Gaia*, provides high-performance storage backends without introducing central trusted parties.

The architecture decouples the security of name registration and name ownership from the availability of data associated with names by separating the control and data planes.

Introduces four layers, with two layers (*blockchain* layer and *virtualchains* layer) in the control plane, and two layers (routing layer and data storage layer) in the data plane.

The control plane consists of a *blockchain* and a logically separate called a *virtualchains*. In the *blockchain* and in the *virtualchains* (Control Plane) is defines the protocol for registering human-readable names, creating (name,hash) bindings (by example: bl.seaco.gc, x7@#34.....) and creating bindings to owning cryptographic keypairs.

Virtualchains are like virtual machines, where a specific VM like Debian 8.7 can run on top of a specific physical machine. Different types of *virtualchains* can be defined and they run on top of the specific underlying blockchain.

Virtualchains operations are encoded in valid *blockchain* transactions as additional meta-data.

Blockchain nodes do see the raw transactions, but the logic to process virtualchain operations only exists at the virtualchain level.

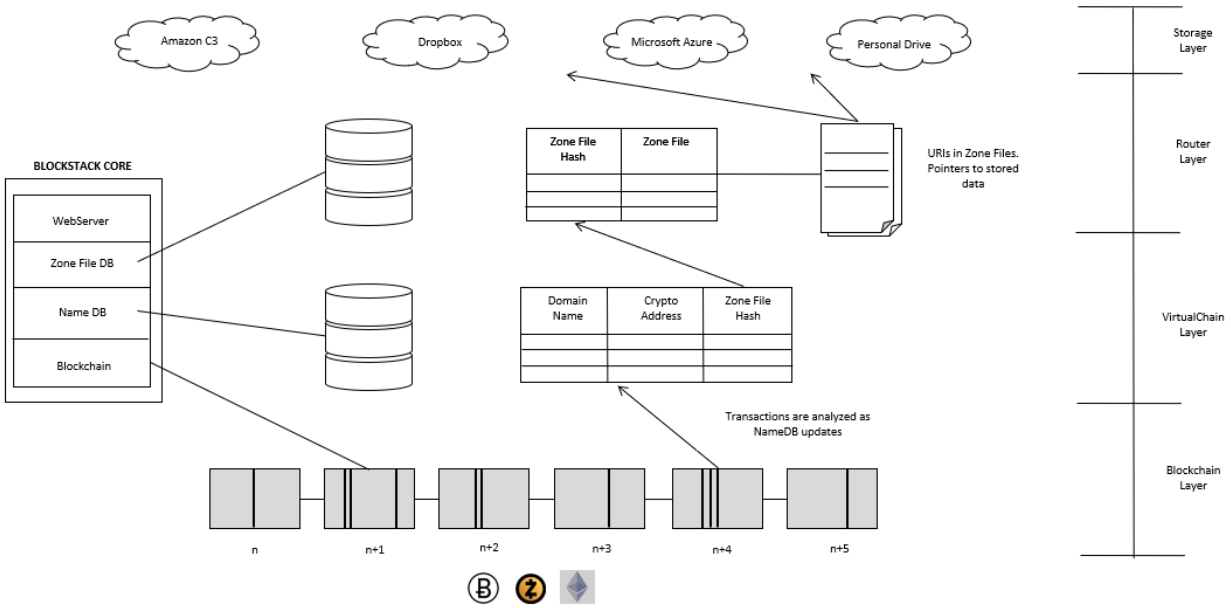
The data plane is responsible for data storage and availability. It consists of:
a) zone files for discovering data by hash or URL, and
b) external storage systems for storing data.

Data values are signed by the public keys of the respective name owners. Clients read data values from the data plane and verify their authenticity by checking that either the data's hash is in the zone file, or the data includes a signature with the name owner's public key.

This hides the individual APIs for storage backends and exposes a simple PUT/GET interface to Blockstack users. Looking up data for a name, like werner.id, works as follows:

a) Lookup the name in the virtualchain to get the (name, hash) pair.
b) Lookup the hash (name) in the Atlas network to get the respective zone file (all peers in the Atlas network have the full replica of all zonefiles).
c) Get the storage backend URI from the zonefile and lookup the URI to connect to the storage backend.
d) Read the data (decrypt it if needed and if you have the access rights) and verify the respective signature or hash.

CONTROL PLANE AND DATA PLANE WITH BLOCKSTACK FRAMEWORK



Blockstack framework

Blockstack proposes to be the new decentralized internet. With Blockstack, users control their data and apps run on their devices. There are no middlemen, no passwords, no massive data silos to breach, and no services tracking us around the internet.

The applications on Blockstack are server-less and decentralized. Developers start by building a single-page application. Then, instead of plugging the frontend into a centralized API, they plug into an API run by the user. Developers install a library called blockstack.js and don't have to worry about running servers, maintaining databases, or building out user management systems.

Personal user APIs ship with the Blockstack app and handle everything from identity and authentication to data storage. Applications can request permissions from users and then gain read and write access to user resources.

Data storage is simple and reliable and uses existing cloud infrastructure. Users connect with their Dropbox, Google Drive, S3, etc.… and data is synced from their local device up to their cloud.

Identity is user-controlled and utilizes the blockchain for secure management of keys, devices and usernames. When users login with apps, they are anonymous by default and use an app-specific key, but their full identity can be revealed and proven at any time. Keys are for signing and encryption and can be changed as devices need to be added or removed.

Under the hood, Blockstack provides a decentralized domain name system (DNS), decentralized public key distribution system, and registry for apps and user identities.

Blockstack contains the following layers:

Layer 1: Blockchain Layer. Occupies the lowest tier, and serves two purposes: it stores the sequence of Blockstack operations and it provides consensus on the order in which the operations were written. Blockstack operations are encoded in transactions on the underlying blockchain.

Layer 2: Virtualchain Layer. Above the blockchain is a virtualchain, which defines new operations without requiring changes to the underlying blockchain. Only Blockstack nodes are aware of this layer and underlying blockchain nodes are agnostic to it. Blockstack operations are defined in the virtualchain layer and are encoded in valid blockchain transactions as additional metadata. Blockchain nodes do see the raw transactions, but the logic to process Blockstack operations only exists at the virtualchain level.

The rules for accepting or rejecting Blockstack operations are also defined in the virtualchain. Accepted operations are processed by the virtualchain to construct a database that stores information on the global state of the system along with state changes at any given blockchain block. Virtualchains can be used to build a variety of state machines.

Layer 3: Routing Layer. Is separates the task of routing requests (i.e., how to discover data) from the actual storage of data. This avoids the need for the system to adopt any particular storage service from the onset, and instead allows multiple storage providers to coexist.

It uses zone files for storing routing information, which are identical to DNS zone files in their format. The virtualchain binds names to respective hash (zone f ile) and stores these bindings in the control plane, whereas the zone files themselves are stored in the routing layer.

Users do not need to trust the routing layer because the integrity of zone files can be verified by checking the hash (zone file) in the control plane.

In the architecture, nodes form a DHT-based peer network for storing zone files. The DHT only stores zone files if hash (zone file) was previously announced in the blockchain.

A distributed hash table (DHT) is a class of a decentralized distributed system that provides a lookup service similar to a hash table: (key, value) pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a

DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

The key aspect relevant of the design is that routes (irrespective of where they are fetched from) can be verified and therefore cannot be tampered with. Further, most production servers maintain a full copy of all zone files since the size of zone files is relatively small (4KB per file). Keeping a full copy of routing data introduces only a marginal storage cost on top of storing the blockchain data.

Layer 4: Storage Layer. The top-most layer is the storage layer, which hosts the actual data values of name-value pairs. All stored data values are signed by the key of the respective owner of a name. By storing data values outside of the blockchain, allows values of arbitrary size and allows for a variety of storage backends.

Users do not need to trust the storage layer because they can verify the integrity of the data values in the control plane.

When we are using a Blockstack client there is control of the data and the ID with a private key. This private key never leaves the device and is meant to stay on the own laptop/phone. As long as no one gets access to the private key, no one can control data or IDs.
Blockstack ensures that all data is signed and verified and (optionally) encrypted end-to-end.

There are two modes of using the storage layer and they differ in how the integrity of data values is verified; the architecture supports both storage modes simultaneously.

a) **Mutable Storage** is the default mode of operation for the storage layer. The user's zone file contains a URI record that points to the data, and the data is constructed to include a signature from the user's private key. Writing the data involves signing and replicating the data (but not the zone file), and reading the data involves fetching the zone file and data, verifying that hash (zone file) matches the hash in Blockstack, and verifying the data's signature with the user's public key. This allows for writes to be as fast as the signature algorithm and underlying storage system allows, since updating the data does not alter the zone file and thus does not require any blockchain transactions. However, readers and writers must employ a data versioning scheme to avoid consuming stale data.

b) **Immutable Storage** is similar to mutable storage, but additionally puts a TXT record in the zone file that contains hash (data). Readers verify data integrity by fetching the data and checking that hash (data) is in the zone file, in addition to verifying the data's signature and the zone file's authenticity. This mode is suitable for data values that don't change often and where it's important to verify that readers see the latest version of the data value. For immutable storage, updates to data values require a new transaction on the underlying blockchain (since the zone file must be modified to include the new hash), making data updates much slower than mutable storage.

A decentralized high-performance storage system with gaia hub works by hosting data in one or more existing storage systems. These storage systems are typically cloud storage systems, or other backend support as well. Gaia enables applications to access to data it via a uniform API.

The Gaia storage system Will be use to store data on behalf of a user. When the user logs in to an application, the authentication process gives the application the URL of a Gaia hub, which performs writes on behalf of that user. The Gaia hub authenticates writes to a location by requiring a valid authentication token, generated by a private key authorized to write at that location.

BILL OF LADING AND SMART CONTRACT WITH KADENAIO FRAMEWORK

The traditional Bill of Lading (BL) will be complemented with a *Smart Contract* on the BlockStack and KadenaIO Pact. Each BL could have a completely different *Smart Contract* as the container cargo could be different. This method will provide a high level of security, assurance of compliance of quality measures, traceability, flexibility and a more efficient process to archive past transactions – all key issues in the logistics industry.

The Bill of Lading is the core of the system and is associated with a Blockstack user (for example proe.id) and a *Smart Contract* that evaluates service compliance. The *Smart Contract* gets executed whenever new measurement information is received evaluating if that information is complying with the rules inicially defined.

In addition, the BL is associated with a container and associated with several communication possibilities such as satellite telephone, mobile telephone, among others (PSTN, GSM, etc ...).

Finally, a BL will also be associated with the IoT devices and sensors.

The system before the association, must verify that the appropiate IoT device is available or occupied.

The user node that assigns the BL with the associations described above (for example proe.id) must top-up BTC in its own application BTC account and in order to operate the system will transfer funds (BTC) to the *Smart Contract* associated with the service. The service requieres funds to operate.

In each container the IoT devices captures the different measurements and encodes them to be sent to the TMS server. The TMS server processes and sends the received data from the IoT devices through a specialized Peer to Peer Network based in Atlas, but modified for this solution, renamed as “Globalconnect network”. The components of this network are the TMS Server and several TMS nodes that could be deployed to cover different territories. The TMS server decodes and sends the data of the measurements in XML or JSON format to the *TMS Node* that is in the Atlas Network of the BlockStack of the *TMS Node* ID.

The TMS node receives the data from the TMS Server in XML or JSON format and performs the following actions:

1) Execute the *Smart Contract* using the received information.
2) Compare the received measurement with the *Smart Contract* rules.
   a. If does NOT comply then:
      - Records the issue in Blockstack (Virtualchain layer and user Storage layer) and in the underlying Blockchain.
      - Send specific user alerts, to be determined (SMS, eMAIL).
   b. If it complies then:
      - Records only in Blockstack (virtualchain layer and user storage layer).

The *Smart Contract* is used to record information. In this way, each time a measure is received, the *Smart Contract* is executed.

From the above it follows that the following information will be stored in the underlying Blockchain:

- Assignment of BL.
- Non-compliance with Smart Contract.
- Termination of the service associated with the BL.

The user node that assigns a IoT device to the BL (for example, proe.id) will have to pay also to the *TMS Node* account a guarantee for returning the of the IoT device, so that it registers that event in the Blockchain.

Once the user who has previously assigned, the service is completed and the IoT device is returned, the *TMS Node* returns the guarantee to the original sender account.

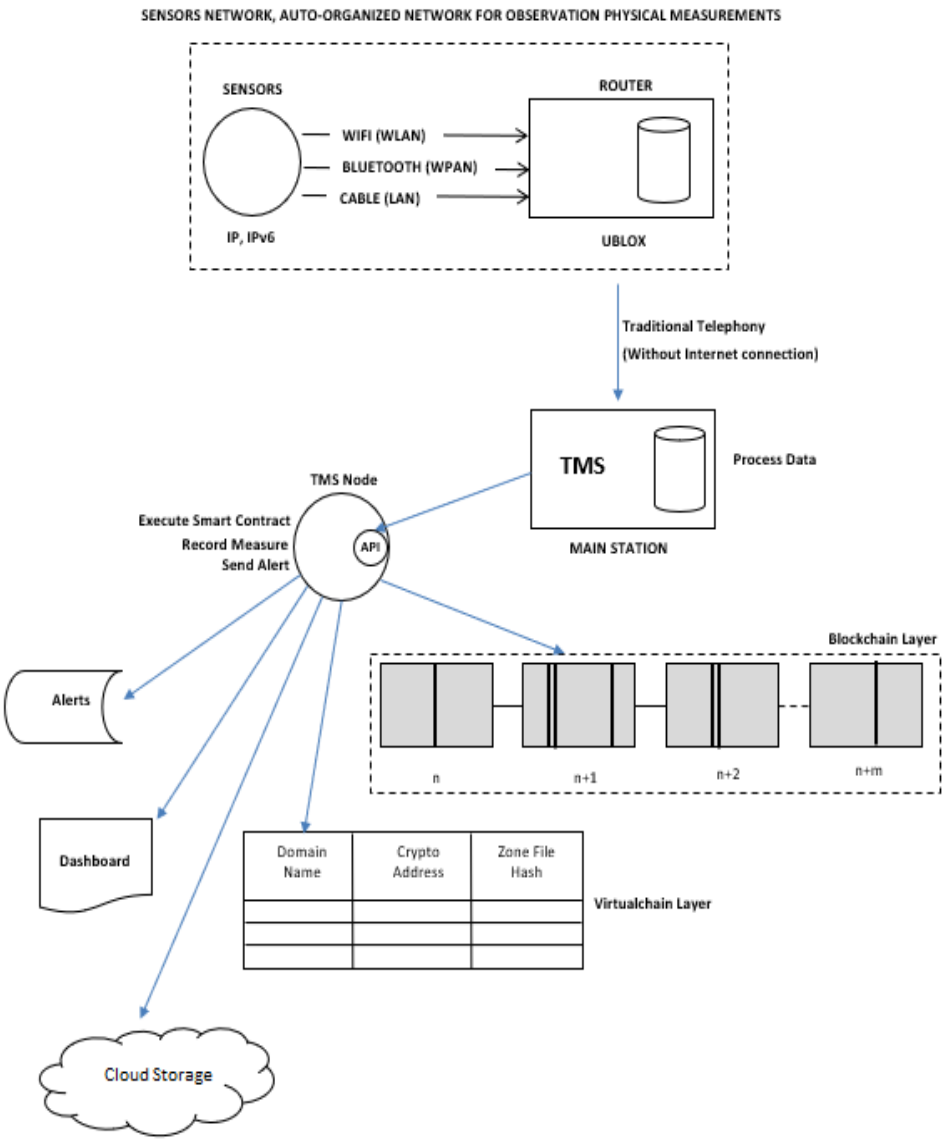
The user who assigns and requests the traceability service authorizes the *TMS Node* to write the storage layer associated with the service. And this same authorizes other users to read the same data.

TMS Node API

The *TMS Node* API contains classes and methods that allow functionalities such as: search user on which the data is stored, apply the rules of the Smart Contract (associated with the BL), determine if it complies or if it does not comply, write measures in Blockchain and/or Blockstack, send alerts via SMS and/or Email.

The parameters needed are:

- Date and time of shipment (by TMS)
- Date and time received by the *TMS Node*
- Caller ID (from where it is send, to look for user)
- Record type (if temperature, humidity, location, etc ...)
- The actual measurement received.


SENSORS NETWORK, AUTO-ORGANIZED NETWORK FOR OBSERVATION PHYSICAL MEASUREMENTS
SENSORS
ROUTER
WIFI (WLAN)
BLUETOOTH (WPAN)
CABLE (LAN)
IP, IPv6
UBLOX
Traditional Telephony
(Without Internet connection)
TMS
Process Data
MAIN STATION
TMS Node
Execute Smart Contract
Record Measure
Send Alert
API
Blockchain Layer
Alerts
n
n+1
n+2
n+m
Dashboard
Domain Name
Crypto Address
Zone File Hash
Virtualchain Layer
Cloud Storage

Workflow