OT-RFC 08

OT Node Blockchain Service Module specification

Proposed by:

Trace Labs - OriginTrail Core Developers

Document authors:

Uroš Kukić (Trace Labs), Miloš Kotlar (Trace Labs), Djordje Kovačević (Trace Labs), Branimir Rakić (Trace Labs)

Versions:

- 2020-12-27 (v1)
- 2020-11-17 (Draft 3)
- 2020-10-09 (Draft 2)
- 2020-09-29 (Draft 1)

Introduction

This document serves as a technical implementation guide for adapting the ot-node to communicate with multiple blockchains during runtime. Before and during implementation this document should serve as a proposal and guideline of the adaptation. After implementing the functionalities, the contents of this document should be adapted to the OriginTrail documentation at docs.origintrial.io.

Problem definition

The current implementation of the OT-node blockchain service is built to support communication with one blockchain at a time. The v5 (Starfleet stage) release of the ot-node requires that this functionality is extended, meaning that a node should be able to communicate with multiple blockchains simultaneously.

Solution proposal

The main focus of this proposal is to improve the decoupling of blockchain features from other parts of the node and move more logic away from specific blockchain implementations and into the blockchain service.

The proposed solution has three main sections:

1. Communication of the blockchain service with specific implementations
2. The internal architecture of the blockchain service
3. Communication of the blockchain service with other node modules and services

After those sections we discuss possible changes when migrating the node to this solution and possible future improvements which arose during development of this proposal.

Communication with implementations

The blockchain service module should serve as a controller between different blockchain implementations, the appropriate blockchain configuration, or multiple ones when required. The service module should be able to communicate with the implementations in two basic ways: listening to events and reading data from the chain.

When receiving data from a blockchain, it is the service's responsibility to add contextual data about the returned values (for example convert to an identity hex value to a DID or add a blockchain_id parameter if the node didn't specify which blockchain to use).

Listening to events

So far the blockchain service has served mostly as a proxy for the actual blockchain implementation, and because of that the node left a lot of control to the Ethereum module. With the introduction of multiple chains at runtime, control should be shifted to the service module, which would then standardize the communication between a blockchain implementation and the rest of the system.
One important aspect of the implementation of the event listeners is to enable each implementation to be refreshed with different intervals and without interdependence. Meaning that a node should not wait for all of the blockchain implementations to fetch events to start handling them, rather it should handle each implementation separately, as soon as it returns the events.

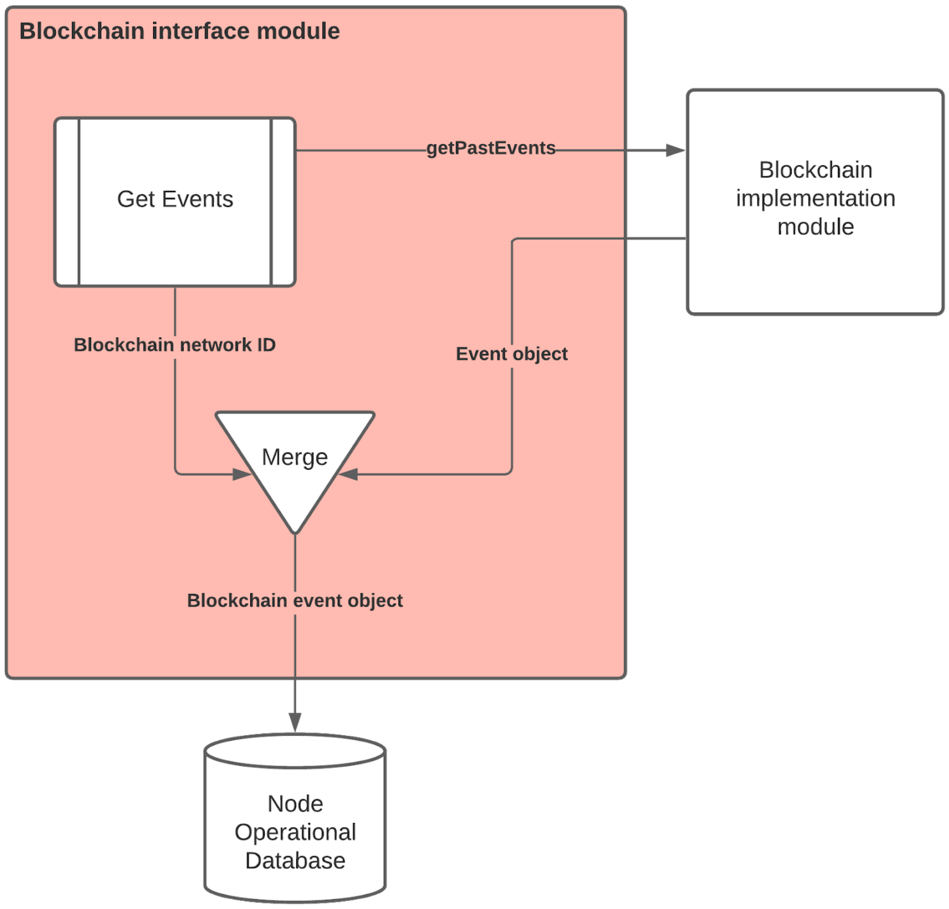
Permanent event listening

A listener is started upon node startup in the form of a periodical function. This function requests events from each blockchain and saves the event data in the node's local database along with additional metadata or editing of identifiers to conform to a standardized format (such as a decentralized identifier).

Every blockchain implementation is required to request all events from its associated contracts between the present and the last timestamp of the event request.
The figure below visualizes how the blockchain service would add context data to blockchain events from the Ethereum blockchain. Other implementations would be implemented in a similar way.

Temporary event listening

A temporary listener is set as a periodical function with a given timeout. The function should check the events table and return the event entry if it exists and return an *undefined* value if event was not found after the timeout expires. It is the caller's responsibility to handle receiving an *undefined* response from the blockchain service.


Blockchain interface module
Get Events
getPastEvents
Blockchain implementation module
Blockchain network ID
Event object
Merge
Blockchain event object
Node Operational Database

Retrieving data / Sending transactions

Simple calls to the blockchain should be straightforward to adapt by just calling the required one or more blockchain implementations and passing the returned value to the caller.
The blockchain service should return the value returned by the blockchain function with two possible alterations:

1. If the value returned is an (decentralized) identifier, the service should convert the value to a DID first.
2. If the function is called for multiple implementations the blockchain service should return an array of objects, with each object containing the blockchain implementation result and a blockchain identifier.

Internal architecture

Configuration

Implementation specific configuration

The blockchain service should have a specific configuration object for each blockchain implementation. This requires adapting the current configuration structure to better reflect which parameters of the node are associated with a specific blockchain implementation (such as Ethereum) and which are general. Most prominently, the node's blockchain identity address and its ethereum keys should be moved to an Ethereum blockchain object. When moved, other modules should not be able to read these parameters directly, but rather we should implement getter functions for reading these parameters from the blockchain service.
While some configuration parameters would be required for every implementation (such as an access/RPC endpoint), there are some parameters that would be used only on some implementations. This means that we need to adapt our tests to better reflect which fields are mandatory and on which blockchain implementations.

Global blockchain configuration

There could be a need for configuration parameters used only by the blockchain service and not by any specific blockchain implementation. To ensure we support this functionality the blockchain configuration should contain all of the blockchain specific configuration parameters in a separate subsection titled "implementations".

Proposed configuration structure

```
{
    ...
    “blockchain”:{
        “default_chain_decision”: “fastest”, // Just an example, do not implement
        ...
        “implementations”: [
            {
                “blockchain_id”: “ethr”,
                “node_wallet”: “0x123fd...”,
                ...
            },
            {
                “blockchain_id”: “sfl”,
                “identity_filepath”: “identities/starfleet.json”,
                ...
            }
            ],
            ...
        },
        ...
}
```

Construction and initialization

Upon starting the blockchain service, the service should be responsible for creating and initializing each blockchain implementation configured in the node configuration.
Blockchain initialization would remain largely unchanged since it is highly specific for each implementation, however errors thrown by any blockchain implementation caught and handled by the blockchain service, meaning that the node should be able to start if some (but not all) blockchain implementations failed to initialize. If all blockchain implementations failed to initialize an error should be thrown to the node.

If some blockchain implementations failed to initialize the node should periodically attempt to reinitialize it.

Identity files and file paths

The blockchain specific identity information (such as identity DID) of a node should be stored in a file so it can be loaded upon node startup. These identity files should remain separate from each other in order to reduce the risk of data corruption and to simplify data parsing in each module that requires an identity file.
While a user should be able to change the filepath for any specific blockchain implementation, we should adhere to some guidelines on the default file path structure. We propose the format *“identities/<blockchain_identitfier>.json”*. This option would cause less clutter in a node’s data folder but maybe require adapting the node backup and restore scripts to handle all the possible scenarios with the identity folder.
When a node first provisions its decentralized identity on a specific chain, the profile service should request the blockchain service to store the identity document information in an identity file. The service should forward this request to the specified blockchain implementation and it should store the identity in an appropriate file.

Decentralized Identifier Adapter

By itself a blockchain implementation does not know about decentralized identifiers. For example, the Ethereum implementation uses hex values of either 20 or 32 bytes. However, DIDs are an upcoming standard which will be used by node modules when calling the blockchain service.
The blockchain module should have a mechanism to convert values from pure hex form to a decentralized identifier and vice versa. This mechanism would be used in both directions, resolving the DID when sending data to the blockchain and associating the appropriate DID prefix (method) when receiving “bare” hex values from the blockchain implementation.

Initially the two DIDs OT-node should support are for Ethereum and Starfleet network (did:ethr and future Starfleet DID method), but we should make sure that other DID conversions would be easy to include in the future, with did:sfl being added upon implementation of the new blockchain.

Because DIDs can be used outside the scope of blockchain implementations, the adapter shouldn’t be a submodule of the blockchain service so it can be easily used for other methods of communication, such as our peer-to-peer network.

Communication with node modules

The blockchain service will receive calls to blockchain functions and return the appropriate response.
Every blockchain call may contain a blockchain identifier to specify which blockchain implementation to use, but the service should have a default choice in the appropriate cases (such as creating an offer).

General function calls

General getters that require data from every blockchain implementation should return an array of values with contextual information on which chain does each value come from. An example response is shown below.

This response structure should also be used in functions which return a single value but could be used without a blockchain identifier to enable the caller to read the blockchain_id from the response.

```
[
    {
        “blockchain_id”: “ethereum”,
        “response”: { ... } // NOTE: This could be an object but also a promise
    }
    {
        “blockchain_id”: “starfleet”,
        “response”: { ... }
    }
]
```

When receiving this kind of response the node should be able to handle all responses independently, removing potential delays caused by one blockchain implementation being slower than another.

Specific function calls

If a network identifier is specified, the blockchain call should behave similarly to the general function call, returning an object containing the blockchain id and the response from that implementation.
Due to different blockchain implementations working in a different way there might be some functions that are available on some implementations and not available on others. In this case, if a blockchain implementation is specified but a function is not available on that implementation, an error should be thrown with the appropriate message.

Default blockchain resolver

The blockchain service should contain a mechanism that chooses which blockchain implementation to call if only one blockchain should be called and the identifier is not specified. This mechanism can be trivial at first, such as calling the blockchain implementation first mentioned in the node configuration, but it should be left adaptable for more complex implementations in the future.

Using blockchain identities in communication

When a node module reads the node identity values (e.g. sending its ERC-725 blockchain identity over the network when bidding for a new dataset replication) the node reads that identity value directly from the node configuration. With the introduction of multiple blockchain implementations, the node should get this identity value from the profile service rather than the configuration. The blockchain service would be called by the profile service and it will get the identity from the proper blockchain implementation and return it to the caller module.

Migration of the blockchain module from OT-Node v4 to v5 (Starfleet stage)

Contextualizing blockchain data

In order to transition to a multi chain supporting node, the current node needs to adapt all of the blockchain related data currently stored on it to contain a blockchain ID, so that when there is data available from another blockchain it has the necessary context to distinguish between them.
The most common way to do this would be to add a column in a node's database which would contain a **blockchain identifier**. Adding blockchain data to a dataset so its fingerprint can be verified without an accompanying blockchain identifier is an ongoing question still to be resolved.

Handling v4 configuration structure

With the changes proposed for the configuration structure, the node should be able to detect that the configuration file has an incorrect configuration structure and warn the user of the possible error.
Implementation details for backwards compatibility with the current configuration structure is still under discussion and to be proposed in the future versions of this document.

Handling v4 dataset structure

With the changes proposed for the dataset structure, the node should be able to detect that the dataset has an older configuration structure and handle the dataset appropriately.
Implementation details for backwards compatibility with the current dataset structure is still under discussion and to be proposed in the future versions of this document.

Data integrity service

The purpose of this service is to ensure the data integrity by signing and verifying the signature of datasets, messages, offer bids, and other content. Nodes should store identity and wallet information for each blockchain implementation. In order to avoid storing sensitive data for multiple blockchain implementations in the same file, each blockchain implementation should store file paths for identities and wallets. Furthermore, this service may implement other methods related to data integrity such as calculating dataset and merkle root hashes, as well as encryption mechanisms.

Proposed configuration structure for global configuration file

```
{
    ...
    “blockchain”:{
        ...
        “implementations”: [
            {
                “blockchain_id”: “ethr”,
                “node_wallet_path”: “...”,
                “identity_filepath”: “...”,
                ...
            },
            ...
        ],
        ...
    },
    ...
}
```

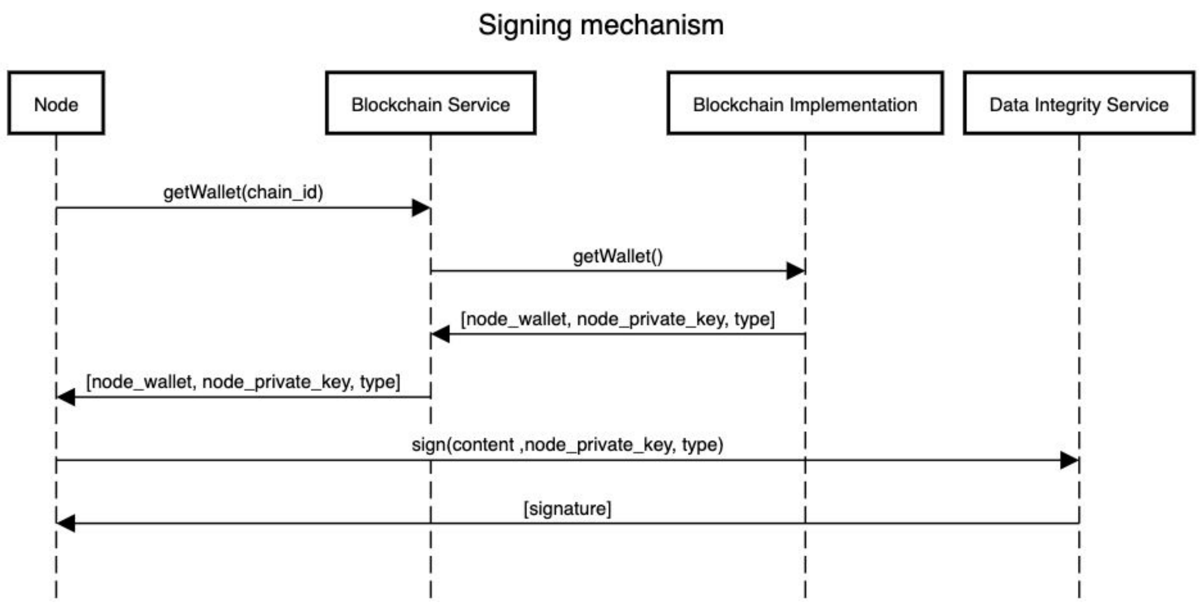
Proposed configuration structure for wallet file

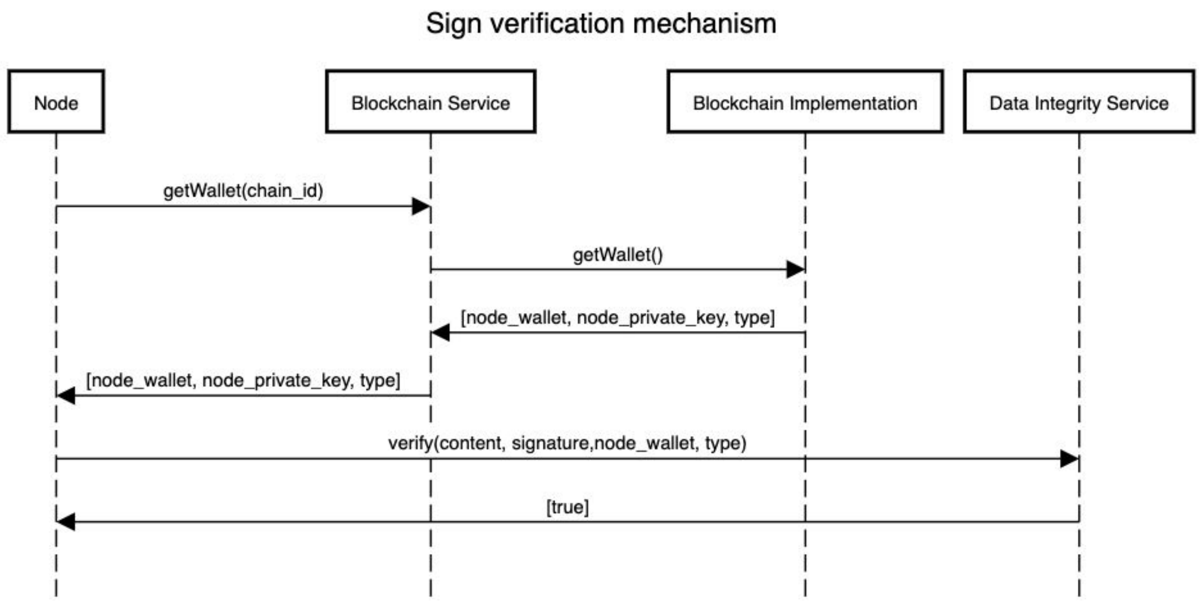
```
{
   “type”: “ECDSA”,
   “node_wallet”:”...”,
   “node_private_key”:”...”,
   “management_wallet”:”...”,
}
```

Proposed configuration structure for identity file

```
{
   “type”:”ERC725”,
   “identity”:”...”,
}
```

Signature sequence diagrams


Signing mechanism
Node
Blockchain Service
Blockchain Implementation
Data Integrity Service
getWallet(chain_id)
getWallet()
[node_wallet, node_private_key, type]
[node_wallet, node_private_key, type]
sign(content ,node_private_key, type)
[signature]


Sign verification mechanism
Node
Blockchain Service
Blockchain Implementation
Data Integrity Service
getWallet(chain_id)
getWallet()
[node_wallet, node_private_key, type]
[node_wallet, node_private_key, type]
verify(content, signature,node_wallet, type)
[true]

Conclusion & Next steps

This document has presented the necessary updates needed to make OT-node capable of interacting with multiple blockchains through it’s blockchain service. Ongoing development will be followed by several milestone releases, first of which will be accompanied with a canary testnet launch. For development specifics please refer to the official OT node github and official documentation.