CRYPTO-SPATIAL : AN OPEN STANDARDS SMART CONTRACTS LIBRARY FOR BUILDING GEOSPATIALLY ENABLED DECENTRALIZED APPLICATIONS ON THE ETHEREUM BLOCKCHAIN

BENAHMED DAHO Ali[1] *

[1] Geodetic Sciences and Topographic Works Engineer, Ain Temouchent, Algeria - bidandou@yahoo.fr

Commission VI, WG VI/6

KEY WORDS: Ethereum Blockchain, Decentralized Applications, Smart Contracts, Ethereum, IPFS, OrbitDB, Land Administration, OGC Open Standards.

ABSTRACT:

Blockchain is an emerging immature technology that disrupt many well established industries. In this contribution we present a smart contracts library, named Crypto-Spatial, written with Solidity for the Ethereum Blockchain and designed to serve as a framework for geospatial data decentrilized permanent storage and retrieviewal.Blockchain is an emerging immature technology that disrupt many well established industries. In this contribution we present a smart contracts library, named Crypto-Spatial, written with Solidity for the Ethereum Blockchain and designed to serve as a framework for geospatial data decentrilized permanent storage and retrieviewal.

1. INTRODUCTION

Situation : Geospatial tech and data layers This project is challenging because it implements geospatial data management features, needed to handle land parcels, on the Blockchain technology, which is an open research subject at the [Open Geospatial Consortium](http://docs.opengeospatial.org/dp/18-041r1/18-041r1.html) where a [Blockchain and Distributed Ledger Technologies Domain Working Group](https://www.opengeospatial.org/projects/groups/bdltdwg) has been created especially for that.

Complications : Decentralization / Blockchain / market growth

Question : Blockchain for geospatial

2. DECENTRALIZED APPLICATIONS

2.1 Ethereum blockchain

Ethereum blockchain can be viewed as a transaction-based state machine: we begin with a genesis state and incrementally execute transactions to morph it into some final state. It is this final state which we accept as the canonical "version" of the world of Ethereum. The state can include such information as account balances, reputations, trust arrangements, data pertaining to information of the physical world; in short, anything that can currently be represented by a computer is admissible. Transactions thus represent a valid arc between two states; the 'valid' part is important. A valid state transition is one which comes about through a transaction. Formally:

$$\boldsymbol{\sigma}_{t+1} \equiv \Upsilon(\boldsymbol{\sigma}_t, T) \tag{1}$$

where Υ is the Ethereum state transition function. In Ethereum, Υ, together with $\boldsymbol{\sigma}$ are considerably more powerful than any existing comparable system; Υ allows components to carry out arbitrary computation, while $\boldsymbol{\sigma}$ allows components to store arbitrary state between transactions.

Transactions are collated into blocks; blocks are chained together using a cryptographic hash as a means of reference. Blocks function as a journal, recording a series of transactions together with the previous block and an identifier for the final state. They also punctuate the transaction series with incentives for nodes to *mine*. This incentivisation takes place as a state-transition function, adding value to a nominated account. Formally, we expand to:

$$\begin{aligned}
\boldsymbol{\sigma}_{t+1} &\equiv \Pi(\boldsymbol{\sigma}_t, B) && (2)\\
B &\equiv (..., (T_0, T_1, ...), ...) && (3)\\
\Pi(\boldsymbol{\sigma}, B) &\equiv \Omega(B, \Upsilon(\Upsilon(\boldsymbol{\sigma}, T_0), T_1)...) && (4)
\end{aligned}$$

Where Ω is the block-finalisation state transition function; B

*Corresponding author

is this block, which includes a series of transactions amongst some other components; and Π is the block-level state-transition function.

This is the basis of the blockchain paradigm, a model that forms the backbone of not only Ethereum, but all decentralised consensus-based transaction systems to date.

In term of implementation, there are many choices of blockchains: over 200 Bitcoin variants, Ethereum and other permissioned blockchains. To meaningfully compare them, [Ji Wang and al.] identified four abstraction layers found in all of these systems. (1) The consensus layer contains protocols via which a block is considered appended to the blockchain. (2) The data layer contains the structure, content and operations on the blockchain data. (3) The execution layer includes details of the runtime environment support blockchain operations. Finally, (4) the application layer includes classes of blockchain applications.

The Crypto-Spatial framework, described in this contribution, is designed for the Ethereum Blockchain and propose a set of smart contracts for the execution layer and a cheap geometry storage solution on IPFS for the application layer.

2.2 IPFS and OrbitDB

IPFS is a distributed file system which synthesizes successful ideas from many peer-to-peer sytems, including DHTs, BitTorrent, Git, and SFS. The contribution of IPFS is simplifying, evolving, and connecting proven techniques into a single cohesive system, greater than the sum of its parts. IPFS presents a new platform for writing and deploying applications, and a new system for distributing and versioning large data. IPFS could even evolve the web itself. IPFS is peer-to-peer; no nodes are privileged. IPFS nodes store IPFS objects in local storage. Nodes connect to each other and transfer objects. These objects represent files and other data structures. [Benet, J. (2014). IPFS-content addressed, versioned, P2P file system. arXiv preprint arXiv:1407.3561.]

OrbitDB. It is a distributed, peer-to-peer database that is built on top of IPFS [60]. OrbitDB supports various kinds of databases including key-value and log databases. This makes OrbitDB an excellent choice for the decentralized prototype.

In the solution we present in this contribution, the geometry of spatial features are stored in GlobalDB as GeoJSON objects for simple parsing and visualisation. The databases also have listeners implemented that triggers when the databases are replicating. Thereafter, the listeners trigger the user interface to update. This ensures that the users will always have the most recent geometry available. [OrbitDB. OrbitDB. url: https://github.com/orbitdb/orbit-db/].

2.3 Decentralized applications developpement

Ethereum blockchain can be viewed as a transaction-based state machine: we begin with a genesis state and incrementally execute transactions to morph it i

Add process and explanation

3. THE CRYPTO-SPATIAL FRAMEWORK

The Crypto-Spatial Framework is a set of smart contracts written in Solidity (programming language for Ethereum) which serves as base classes, like in Object Oriented Programming, that can be inherited and specilized to meet busienss needs. The architecture of the Crypto-Spatial smart constracts is inspired from OGC Simple Features specifications.

To uniquely identify spatial features on the distributed ledger the FOAM space [foam.space] concept of CryptoSpatial Coordinate (CSC) was used. Nevertheless, some modifications was implemented to explore the alternatives suggested by the [OGC discussion paper (7.5)-18-041r1.html] as the use of the H3 javascript library [uber.github.io/h3/], with the resolution 15 (Average Hexagon Edge Length of 0.5 km), which it is a partially conforming implementation of the [Geodesic Discrete Global Grid Systems-http://webpages.sou.edu/ sahrk/sqspc/pubs/gdggs03.pdf) OGC standard.

3.1 Framework Design

The Crypto-Spatial smart contracts library, illustrated in the following class diagram, is designed and implemented using inheritance and interfaces to simplify its resusability.

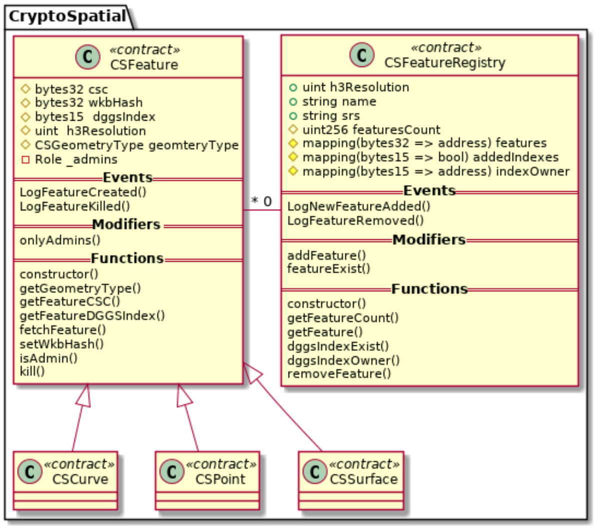

Figure 1. Crypto-Spatial Library Class Diagram

The main goal of this project is to deliver a framework of geospatially enabled smart contracts and libraries for secure GeodApps developement.

1. The developper should be able to use a deployed ‘geospatial‘ smart contracts and libraries that suit his business needs from ENS (ens.domains) resolvable Ethereum addresses.

1. The developper should be able to inherit/use the Solidity components to build his custom contracts and develop more complex decentralized systems.

1. The developper should be able to integrate/donwload the solidity reusable components (contacts) from [npm](https://www.npmjs.com/) and/or [ethPM](https://www.ethpm.com/) as standalone packages

or included in a widely accepted library, like [openZeppelin](https://openzeppelin.com/contracts/).

1. The developper should be able to visualize on a map all the gospatial features stored on the permanently deployed registry (as features index) to administer the features belonging to his regitries.

1. The developper should be able to access a fully featured dasboard displaying all useful information about the permanently deployed features registries (the features index).

1. The developper should be able to interact with the features index with the [OGC OpenAPI REST API](http://docs.opengeospatial.org/wp/16-019r4/16-019r4.html) to discover and fetch content.

3.1.1 CSFeature The library design, inspired from OGC Simple Features, comprises a base abstract CSFeature smart contract to represent any type of spatial features. This smart contract is specialized to hanldle Points, Curves and Surfaces with the CSCurve, CSPoint and the CSSurface smart contracts respectively.

The CSFeature smart contract includes all necessary state variables, modifiers, events and functions to store and manipulate spatial features. Formally :

csc : the Crypto-Spatial Coordinate, which is the Keccak-256 hash of the DGGS index of the spatial feature and the owner address.

wkbHash : the Well Known Binary Hash of the spatial feature GeoJSON geometry.

dggsIndex : the Disrcete Global Geodetic System index of the spatial feature.

h3Resolution : the H3 resolution used for the spatial feature.

CSGeometryType : the geometry type of the spatial feature (Point, Curve, Surface).

constructor : the constructor that initiate all state variables.

getGeometryType : getter for geometry type.

getFeatureCSC : getter for CSC.

getFeatureDGGSIndex : getter for DGGS Index.

fetchFeature : fetch all the state variables of the feature

setWkbHash : setter for wkbHash.

kill : to permanently remove the spatial feature from the blockchain ledger.

3.1.2 CSFeatureRegistry The second important abstract smart contract of the Crypto-Spatial core library design is the CSFeatureRegistry which serves as the spatial features collection. The CSFeatureRegistry smart contract includes all necessary state variables, modifiers, events and functions to store and manipulate spatial features. Formally :

h3Resolution : the H3 resolution of the spatial feature registry (from 1 to 15) see [.....].

name : the displayed name.

srs : the Spatial Reference System code (EPSG or equivalent).

featuresCount : the Counter of the added features.

features : an addresses mapping to handle the spatial features added to the registry.

addedIndexes : a boolean mapping to keep trace of added indexes.

indexOwner : a mapping to keep trace of DGGS indexes owners.

constructor : the constructor that initiate all state variables.

addFeature : a modifier that must be called by the addFeature function of inherited smart contract.

getFeatureCount : getter of the featureCount.

getFeature : getter for a designated spatial feature.

dggsIndexExist : to confirm if an Index exist in the registry.

dggsIndexOwner : returns the owner of a designated feature.

removeFeature : permanently remove the spatial fetaure from the registry and the blockchain ledger.

3.2 Solidity Implementation

To demonstrate the suitability of the previous design, the smart contracts laibrary has been implemented using the Solidity programming language and the Truffle suite. Herafter, a code snipet of the addFeature modifier demonstrating the ability to resuse business logic on the Etheruem smart contracts :

```
modifier
addFeature(bytes15 dggsIndex,
   bytes32 wkbHash,
   address _sender) {
  require(!paused(), "Contract is paused");
  require(dggsIndex.length != 0, "Empty dggsIndex");
  require(addedIndexes[dggsIndex] == false,
   "DGSS Index already exist");
  require(wkbHash.length != 0, "Empty wkbHash");

  _;

  addedIndexes[dggsIndex] = true;
  indexOwner[dggsIndex] = _sender;
  bytes32 csc = CSGeometryLib.computeCSCIndex(_sender,
   dggsIndex);
  emit LogNewFeatureAdded(name, csc, dggsIndex,
   wkbHash, _sender);
  featuresCount = featuresCount.add(1);
}
```

The complete implementation of the Crypto-Spatial framework can be found on the following github repository. https://github.com/allilou/onchain-land-administration/tree/master/solidity/contracts/geospatial

3.3 Security and design patterns

In this section we systematize the security vulnerabilities of Ethereum smart contracts. We group the vulnerabilities in three classes, according to the level where they are introduced (Solidity, EVM bytecode, or blockchain). Further, we illustrate each vulnerability at the Solidity level through a small piece of code. All these vulnerabilities can be (actually, most of them have been) exploited to carry on attacks which e.g. steal money from contracts. Table 1 summarizes our taxonomy of vulnerabilities, with links to the attacks illustrated in Section 4.

3.3.1 Security issues mitigation As smart contracts are immutable code taht execute in decentralized paradigm a set of strategies to avoid common attacks has to be used. [https://blog.sigmaprime.io/solidity-security.html)

rithmetic Over/Under Flows To guard against under/overflow vulnerabilities we use the openZeppelin “‘SafeMath“‘ mathematical library for the state variable “‘featuresCount (uint256)“‘ in the [CSFeatureRegistry.sol](../solidity/contracts/geospatial/CSFeatureRegistry.sol).

Security tools MythX Having some trouble using the command line analysis ‘truffle run verify‘, the MythX Visual Studio Code was used instead. The full report highlight two low severity issues described bellow.

Detected Issues

— 0 High — 0 Medium — 2 Low — ————————–——
–—

— ID — Severity — Name — File — Location — ——-
———–———–——————–— —SWC-103— Low—
Floating Pragma —CSFeatureRegistry.sol —L: 9 C: 0
— —SWC-108 —Low—State Variable Default Visibility—CSFeatureRegistry.sol —L: 30 C: 10—

Mythril Knowing that the free version of MythX don’t analyze all the security vulnerabilities, the ‘mythril‘ tool was also used to check the smart contracts of this project.

Unfortunately, the latest version (v0.21.20) of this tool don’t returns any analysis result. “‘ mythril always retuns The analysis was completed successfully. No issues were detected. “‘ We tried with docker (on Debian Linux / windows 10) and with the python installed version of mythril, we get the same result !!!. The log was : “‘

Desperate from using ‘mythril‘, the ‘slither‘ tool was used and returns the following results.

“‘ INFO:Slither:./contracts/LAParcelRegistry.sol analyzed (13 contracts with 40 detectors), 36 result(s) found INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration

“‘

3.3.2 Design patterns As smart contracts are special immutable code executing on the Ethereum blockchain, a number of design patterns has to be applied to guarantee they are correctly prepared for all situations. This include :

Fail early and fail loud In the smart contracts of this project we check the condition required for execution (by using “‘require()“‘) as early as possible in the function body and throws an exception if the condition is not met . Those exeptions are also confirmed by the tests.

Restricting Access The acces to the smart contract functions that change the states is restricted using the openZeppelin library. In fact :

- [CSFeature.sol](../solidity/contracts/geospatial/CSFeature.sol) inherit from “‘Ownable“‘ and use “‘Roles“‘ to limit acces to setters and the “‘kill“‘ function.

- In [CSFeatureRegistry.sol](../solidity/contracts/geospatial/CSFeatureRegistry access to “‘removeFeature“‘ function is also limited to admins declared in the constructor.

In addition, the acces to the state variable of the [CSFeature.sol](../solidity/contracts/geospatial/CSFeature.sol) smart contract is restricted to “‘internal“‘.

Mortal The [CSFeature.sol](../solidity/contracts/geospatial/CSFeature.sol) allow autodestruction which remove it definitly from the blockchain using the “‘removeFeature“‘ function.

Auto Deprecation

The auto deprecation design pattern is not used. We prefere using “‘Pausable“‘.

Circuit Breaker The [CSFeatureRegistry.sol](../solidity/contracts/geospatial/CSFeatureRegistry.sol) inherit from the openZeppelin “‘Pausable“‘ contract which allow pausing the smart contract from adding and removing features from the registry.

Inheritance and interfaces The geospatial library part of this project, illustrated in the following class diagram, is designed and implemented using inheritance and interfaces to simplify its resusability.

4. DECENTRALIZED LAND ADMINISTRATION

Type text single-spaced, **with** one blank line between paragraphs and following headings. Start paragraphs flush with left margin.

4.1 The workflow

DeLA plateform is a Decentrelized Application (dApp) for Land Administration built for the Ethereum blockchain. It comprises 03 components :

1. The smart contracts written in Solidity
2. The IPFS/OrbitDB Storage
3. The frontend web application built with react.js

The land registry and real estate transactions is one area where security and transparency are important and where there is a high level of value, but where the required transaction speed and the number of transactions is significantly lower. People who have looked at this area, like the [Economist](https://www.economist.com/leaders/2015/10/31/the-trust-machine), understand that the value for society may be enormous — not least in countries that lack stable institutions such as legal systems, land registries, etc.

The aim of this project is to implement the [ISO 19152:2012 standard (Geographic information — Land Administration Domain Model (LADM))](https://www.iso.org/standard/51206.html) on the Ethereum Blockchain.

The starting point for that will be the [Solutions for Open Land Administration (SOLA-FAO)](https://github.com/SOLA-FAO/) which is a J2EE implementation that has many uses cases in Africa and Asia. Using SOLA allows us to incorporate international best practice and standards, facilitating the essential customization always required to meet specific country needs, as Cadastre and registration functions and services are always provided by a typical land office.

One of the major goals for porting this land registry solution to the etherum blockchain is the ability to use it as a [crowd sourcing land registry plateform](http://www.fao.org/tenure/voluntary-guidelines/en/) to collect tenure relationships and as a tool for communities to assess and clarify their tenure regimes so to protect the individual and collective rights of their members.

For the frontend, the [Leaflet](https://leafletjs.com/) library is used to display a map with a markers representing the added indexes.

5. BLOCKCHAIN BUSINESS MODELS

The maximum paper length is restricted to 8 pages. Invited papers can be increased to 12 pages.

6. CONCLUSION

The maximum paper length is restricted to 8 pages. Invited papers can be increased to 12 pages.

Porting to Hyperledger fabric

ACKNOWLEDGEMENTS (OPTIONAL)

Acknowledgements of support for the project/paper/author are welcome.

APPENDIX (OPTIONAL)

Any additional supporting data may be appended, provided the paper does not exceed the limits given above.

Revised May 2019