Raport z realizacji kamienia milowego

Etap 1

Projekt NCBR POIR.01.01.01-00-0982/16  
31-10-2017

|  |  |
| --- | --- |
| Dokument |  |
| Projekt | NCBR POIR.01.01.01-00-0982/16 |
| Sygnatura wewnętyrzna |  |
| Status |  |
| Historia zmian | |
|  |  |

[1 Wstęp 5](#_Toc499557849)

[1.1 Opis problemu 5](#_Toc499557850)

[1.2 Cel I zakres pracy 9](#_Toc499557851)

[2 Przegląd istniejących rozwiązań 9](#_Toc499557852)

[3 Architektura proponowanych rozwiązań 12](#_Toc499557853)

[3.1 Cechy technologii Blockchain 12](#_Toc499557854)

[3.1.1 Introduction 12](#_Toc499557855)

[3.1.2 Data Sharing 12](#_Toc499557856)

[3.1.3 Data Model 13](#_Toc499557857)

[3.1.4 Node Agent 13](#_Toc499557858)

[3.1.5 Domain Model 14](#_Toc499557859)

[3.2 Architektura BlockGraph 15](#_Toc499557860)

[3.2.1 Wstęp 15](#_Toc499557861)

[3.2.2 Proponowane rozwiązanie 15](#_Toc499557862)

[3.2.3 Implementacja proponowanych rozwiązań 17](#_Toc499557863)

[3.3 Architektura GraphChain 18](#_Toc499557864)

[3.3.1 Wstęp 18](#_Toc499557865)

[3.3.2 Proponowane rozwiązania 19](#_Toc499557866)

[3.3.3 Opracowanie koncepcji struktury grafu do zrealizowania zabezpieczeń kryptograficznych typowych dla Blockchain. 21](#_Toc499557867)

[3.3.4 Stworzenie funkcjonujących mechanizmów kryptograficznych 22](#_Toc499557868)

[3.3.5 Stworzenie mechanizmów replikacji struktur grafowych 24](#_Toc499557869)

[3.3.6 Stworzenie warstwy dostępowej (HTTP API) dla tak przygotowanej implementacji bazy typu RDF Graph. 24](#_Toc499557870)

[3.3.7 Implementacja proponowanych rozwiązań 24](#_Toc499557871)

[4 Badanie implementacji proponowanych rozwiązań 26](#_Toc499557872)

[4.1 Zakres badań i wybór kryteriów oceny 26](#_Toc499557873)

[4.2 Badanie architektury BlockGraph 26](#_Toc499557874)

[4.2.1 Opis testu 26](#_Toc499557875)

[4.2.2 Złożoność obliczeniowa 26](#_Toc499557876)

[4.2.3 Wykonanie kwerendy 27](#_Toc499557877)

[4.2.4 Podsumowanie 29](#_Toc499557878)

[4.3 Badanie GraphChain 30](#_Toc499557879)

[4.3.1 Wprowadzenie 30](#_Toc499557880)

[4.3.2 Sprzęt i oprogramowanie 31](#_Toc499557881)

[4.3.3 Normalizacja grafu, tworzenie bloku 31](#_Toc499557882)

[4.3.4 Interpretacja 33](#_Toc499557883)

[4.3.5 Zapisywanie trójek w repozytorium 34](#_Toc499557884)

[4.3.6 Interpretacja 35](#_Toc499557885)

[4.3.7 Walidacja łańcucha bloków 35](#_Toc499557886)

[5 Podsumowanie wyników 38](#_Toc499557887)

[5.1 Złożoność obliczeniowa 38](#_Toc499557888)

[5.1.1 Tworzenie repozytorium 38](#_Toc499557889)

[5.1.2 Odczyt danych z repozytorium 39](#_Toc499557890)

[5.2 Realizacja zabezpieczenia strumienia 39](#_Toc499557891)

[5.2.1 BlockGrapH 39](#_Toc499557892)

[5.2.2 GraphChain 39](#_Toc499557893)

[6 Wnioski końcowe 40](#_Toc499557894)

[6.1 Wstęp 40](#_Toc499557895)

[6.2 Bezpieczeństwo danych 41](#_Toc499557896)

[6.3 Wydajność 42](#_Toc499557897)

[6.4 Elastyczność 42](#_Toc499557898)

[7 Podsumowanie 43](#_Toc499557899)

# Wstęp

## Opis problemu

Business Intelligence (BI) comprises the strategies and technologies used by enterprises for data analysis with the goal of providing historical, current, and predictive views of business operations. BI technologies can handle large amounts of data to help identify, develop and otherwise create new strategic business opportunities. Business intelligence based on ontology deployed using triple-store (RDF store) as a storage and retrieval system of data using semantic queries is widely used for organization operation optimization. We can call this approach interactive analysis, the user actively polls a data source (data at rest) for more information.

On the other hand, business may be recognized as a stream of geographically distributed transactions. It means that for each transaction there must be exactly one before and one after except the last one. Even if two not colliding transactions are concurrent they can be arbitrary scheduled in time to impose sequence. We can represent these transactions as a read-only chain-distributed ledger. Blockchain is the world's leading technology and platform to deploy the distributed ledgers. The main goal is to make this chain consistent and trustworthy. As the result the chain is continuously growing (data in transit) so instead of static pooling the user is offered more information by subscribing to a data stream. We can call this approach reactive analysis, because the user is passive in the data retrieval process: apart from subscribing to the observable chain of transactions, it does not actively poll the source, but merely reacts to the data being pushed to it.

In this paper a case study presenting how organizations legal and financial information may be surrounded by important business information to make a one whole offering the same level of data consistency and nonrepudiation as the blockchain does and triple-store searchability by integration both the technologies with the goal of obtaining a homogenous globally scoped solution.

Legal Entity Identifier (LEI) is an example of legal information that issuance procedure can be recognized as the transaction and related data appropriately protected to be read only and no repudiated forever as it is organizations digital asset. For implementation and use of the LEI the [Global Legal Entity Identifier Foundation (GLEIF)](https://www.gleif.org/en/about/this-is-gleif) a not-for-profit organization created is responsible. GLEIF services ensure the operational integrity of the Global LEI System. GLEIF is overseen by the [LEI Regulatory Oversight Committee](https://www.gleif.org/en/about/governance/lei-regulatory-oversight-committee-lei-roc), which is made up of representatives of public authorities from around the globe. LEI issuers are the organizations authorized to issue LEIs to legal entities engaging in financial transactions. – are the organizations authorized to issue LEIs to legal entities engaging in financial transactions.

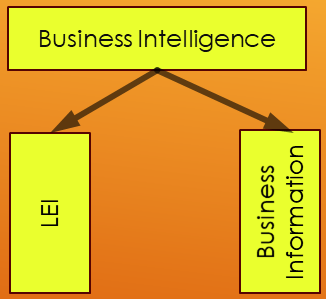


Figure 1 GLEIF side-by side business model

In general the GLEIF business model is presented in Figure 1 where LEI catalog and business information are published, collected, maintained and processed side-by-side by the upper layer. We propose a new embedded model (see. Figure 2) where the legal information is embedded into a selected business data. An example of data where this “carving in stones” data storage model is applicable is financial information represented as an XBRL (eXtensible Business Reporting Language) documents. XBRL is a freely available and global standard for exchanging business information. One use of XBRL is to define and exchange financial information, such as a financial statement.

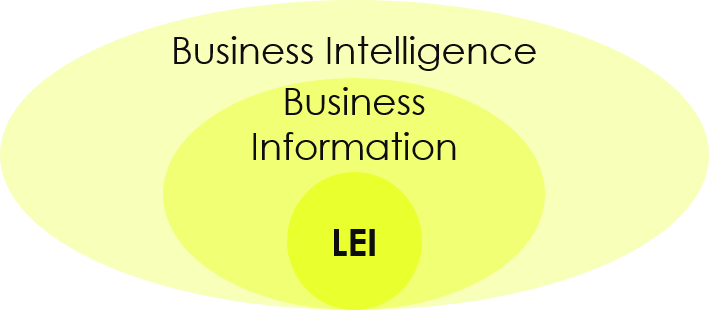


Figure GLEIF embedded business model

To make the proposed approach feasible the interactive and reactive data processing models must be converged. The main challenge faced up by is how to integrate mentioned above blockchain and triple store technologies. The following approaches are considered:

* BlockGraph: Embodying RDF graphs into a chain of blocks and make the blockchain searchable;
* GraphChain: Expanding triple-store to embed historical information and protect selected business information against any changes;

The proposed architecture is presented in Figure 3 as the domain model. Each solution requires requirements analysis, which reveals a number of business terms that must be defined if the requirements are to be understood and clarified. The domain model (business object model) is a powerful mechanism for describing the important terms of the business providing a single definition of the terms and their relationships. One of the advantages of using a domain model is that the terms are modeled as elements, allowing them to be linked to other elements within the domain model itself or to elements in other parts of the models.

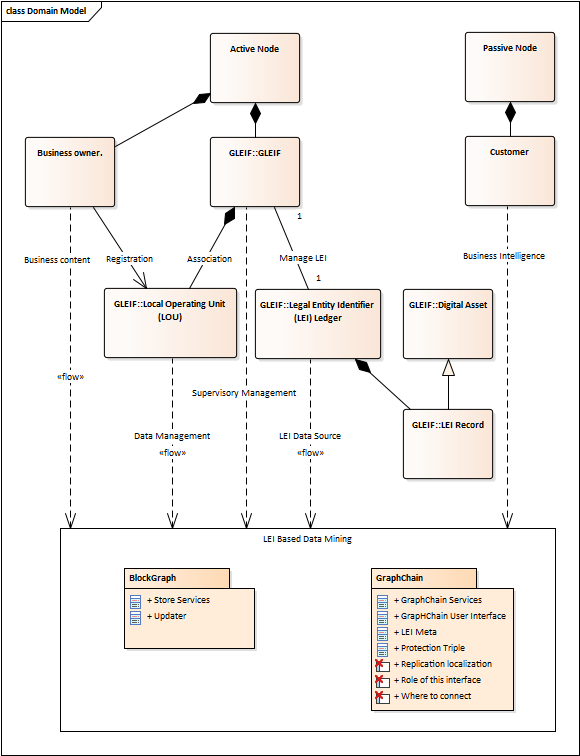


Figure 3 Integration domain model.

The model has been designed on the following assumptions:

* The solution conforms to the distributed ledger (also called shared ledger) concept/technology (DLT) that is a consensus of replicated, shared, and synchronized digital data geographically spread across multiple sites, countries, or institutions.
* The collected already read-only data is exposed to users as the triplestore that allows storage and retrieval of triples through semantic queries.

Implementation of the DLT concept requires the following functionality:

* Factoring of the cryptographically protected against any modification chain of blocks (linked list) that can increase its size by adding new blokes at the end of the chain
* Implementation of a selected consensus algorithm imposing addition of only valid and approved blocks that cannot be modified subsequently;
* Replication mechanism distributing any changes to all nodes on the network that are shared by different clients thanks to the peer-to-peer (P2P) architecture.

Since cryptographic tools are used, it is not possible to modify data already added to the chain. The data will be stored in the chain and be public forever, so this process resembles carving in stones activity.

Triplestore is a database that stores data in context of semantics metadata and retrieves it via a query language. A triplestore is optimized for the storage and retrieval of triples. A triple is a data entity composed of subject-predicate-object.

All the users of the proposed solution (Figure 3) is grouped in the two categories:

* Active – allowed to modify the store
* Passive – retrieving only the data from the store

Passive users are a collection of customers using a query language to get answers for question addressing daily business issues.

One of the active user is an authority responsible to collect, maintain and publish legal or financial information. In the presented case GLEIF organization is mentioned as an example, which meets assumptions described above. In the presented model GLEIF is associated with the role of supervisory management of the LEI ledger implemented inside the proposed system LEI Based Data Mining. The LEI data published in the systems is retrieved from redundant LEI Ledger organized as set of LEI Record’s.

The mining may be recognized as the computing process of discovering patterns in large data sets involving methods at the intersection of machine learning, statistics, and database systems with the overall goal to extract information from a data set and transform it into an understandable structure for further use.

To populate system with the relevant data the Business Owner role has been distinguished. It is assumed that in the proposed model this kind of actor is registered in one of Local Operational Unit organizations and responsible to provide LEI related information. Subsequently this information may be expanded to provide additional semantics aware context where the LEI record is exposed publicly and maintained.

## Cel I zakres pracy

Główny celem tego projektu jest analiza wariantów i wybór rozwiązania, które spełni poniższe wymagania.

1. Przegląd mechanizmów implementacji baz typu RDF Graph ze szczególnym uwzględnieniem „Native Triple Stores” – wybór najbardziej efektywnego mechanizmu.
2. Przegląd istniejących implementacji Blockchain pod kątem stosowalności ich mechanizmów do bardziej uniwersalnych struktur danych.
3. Opracowanie koncepcji struktury grafu zawierającej dodatkowe elementy niezbędne do zrealizowania zabezpieczeń kryptograficznych typowych dla Blockchain.
4. Stworzenie funkcjonujących mechanizmów kryptograficznych Blockchain stosowalnych do struktur grafowych.
5. Stworzenie funkcjonujących mechanizmów replikacji struktur grafowych do innych węzłów systemu.
6. Stworzenie warstwy dostępowej (API) dla tak przygotowanej implementacji bazy typu RDF Graph.
7. Stworzenie oprogramowania testowego i przeprowadzenie testów.
8. Umieszczenie kodu rozwiązania w przestrzeni Open Source.
9. Opracowanie publikacji opisującej badania i otwierającej publiczną dyskusję nad jej wynikami.
10. Opracowanie koncepcji „zanurzenia” reprezentacji grafu (RDF Graph) w „płaskim” bloku danych Blockchain.
11. Opracowanie koncepcji mechanizmów dostępu do grafu zanurzonego w „płaskich” blokach danych.
12. Stworzenie funkcjonujących mechanizmów dokonującego konwersji (upakowania) grafu (RDF w „płaskim” bloku danych – składnia i semantyka bloków BC).
13. Stworzenie funkcjonujących mechanizmów realizującego dostęp do elementów (nodów) grafu zanurzonych w „płaskim” bloku danych.
14. Stworzenie warstwy dostępowej (API) dla tak przygotowanej implementacji bazy grafowej.
15. Stworzenie oprogramowania testowego i przeprowadzenie testów.
16. Umieszczenie kodu rozwiązania w przestrzeni Open Source.
17. Opracowanie publikacji opisującej badania i otwierającej publiczną dyskusję nad jej wynikami.

# Przegląd istniejących rozwiązań

Aktualnie na świecie stosuje się wiele rozwiązań, które tylko częściowo mogą zaspokoić wymagania stawiane przez zastosowania opisane w poprzednim rozdziale. Na podstawie analizy wybranych cech funkcjonalnych, popularności i poziomu gotowości (ang. TRL - Technology Reediness Level) w tabeli poniżej zamieszczono komponenty, które mogą być wykorzystane do dalszych badań.

|  |  |  |  |
| --- | --- | --- | --- |
| Komponent | Funkcja | technologia | Opis |
| dotNetRDF | triplestore | C# | dotNetRDF is a powerful and flexible API for working with RDF and SPARQL in .Net environments dotNetRDF  dotNetRDF is a powerful and flexible API for working with RDF and SPARQL in .Net environments, for details and documentation on the project please see our website at dotnetrdf.org |
| Nethereum | Ethereum | C# | Bringing the love of Ethereum to .Net  Nethereum is the .Net integration library for Ethereum, it allows you to interact with Ethereum clients like geth, eth or parity using RPC. The library has very similar functionality as the Javascript Etherum Web3 RPC Client Library. All the JSON RPC/IPC methods are implemented as they appear in new versions of the clients. The geth client is the one that is closely supported and tested, including its management extensions for admin, personal, debugging, miner. Interaction with contracts has been simplified for deployment, function calling, transaction and event filtering and decoding of topics. The library has been tested in all the platforms .Net Core, Mono, Linux, iOS, Android, Raspberry PI, Xbox and of course Windows |
| CSharp2nem | Nem BlockChain | C# | The CSharp2nem Api wrapper library is designed to simplify development on and use of the nem blockchain. CSharp2nem flexibly supports mainnet, testnet and mijin. |
| NStratis | BlockChain | C# | NStratis is a fork of [NBitcoin](https://github.com/MetacoSA/NBitcoin) with additional support for:  POS (Proof of Stake) mining algorithm  Hash functions for the X13 POW mining algorithm  NStratis is targeted to the Stratis blockchain . It's the core component of the Stratis [FullNode](https://github.com/stratisproject/StratisBitcoinFullNode) framework, a port of the bitcoin blockchain in C# |
| web3j | Ethereum |  | web3j is a lightweight, reactive, type safe Java and Android library for integrating with clients (nodes) on the Ethereum network |
| RDF4J | Triplestore | Java | RDF4J udostępnia programowalne API, za pomocą którego można tworzyć modele RDF, dodawać i usuwać trójki, zapisywać do pliku i odczytywać (wspierane są różne formaty: RDF/XML, Turtle, N-Triples, Json-LD; cała dostępna lista tutaj). Przykłady użycia API można znaleźć pod tym adresem. |
| Apache Jena |  |  | Apache Jena udostępnia programowalne API w Javie. Za pomocą API można tworzyć i modyfikować modele RDF, wczytywać je z plików i zapisywać do plików. Obsługiwane serializacje RDF można znaleźć w tutaj. Przykłady użycia API znajdują się w dokumentacji projektu oraz w innym naszym projekcie: SearchInsights.  Jena dostarcza również natywnego TS, który nazywa się TDB. Oprócz TDB, istnieje również możliwość użycia Fuseki jako serwera SPARQL, ale nie jestem pewien czy można komunikować się z nim bezpośrednio za pomocą programowalnego API (raczej nie). |
| naivechain |  |  | naivechain to implementacja prostego BC w języku JavaScript, wykorzystująca framework node.js, zawarta w ok. 200 liniach kodu. Nie jest to biblioteka do tworzenia BC, ale gotowa, choć bardzo prosta i ograniczona, implementacja BC. naivechain działa jako serwer HTTP oraz udostępnia web service'y do połączeń typu P2P pomiędzy poszczególnymi węzłami. Najważniejsze cechy tej implementacji to: |
| Ethereum | Ethereum | C++ | It is the third most popular of the Ethereum clients, behind geth (the go client) and Parity (the rust client). The code is exceptionally portable and has been used successfully on a very broad range of operating systems and hardware. cpp-ethereum |
| HDT | binary serialization format | C++ | HDT (Header, Dictionary, Triples) is a compact data structure and binary serialization format for RDF that keeps big datasets compressed to save space while maintaining search and browse operations without prior decompression. HDT |
| Redland | triplestore | C | Redland is a set of free software C libraries that provide support for the Resource Description Framework (RDF). |

# Architektura proponowanych rozwiązań

## Cechy technologii Blockchain

### Introduction

The blockchain is recognized as a technology aimed at managing a cryptocurrency and solve the double spending problem. In principle the technology offers creation-to-ends financial transactions protection where a transaction is frozen after having been accepted by a community, i.e. it is made available by a peer-to-peer network in a context, which makes it read only over its entire life-cycle. It is designed to defeat any attempts at modifying because no party including the originator can change the data being maintained by the network of nodes. It makes this technology a perfect option to publish and protect any irrevocable information, e.g. identifiers, financial reports, certificates, privileges, licenses, etc.

### Data Sharing

According to the blockchain technology the protected data is managed and shared by a community of nodes making up a decentralized system, where lower level components operate on local data to accomplish global goals. The local data must be obtained from other nodes. Because the nodes are interconnected and make up a peer-to-peer network they do not need to be identified, i.e. messages containing copy of the shared data are not routed to any particular place and only need to be delivered on a best effort basis. Additionally nodes can leave and rejoin the network at will without degradation of data availability and protection level.

Privileged nodes can modify the shared data by requesting addition of a new data. Because the system is decentralized it must be designed to be Bysantine fault tolerant (BFT). The node tracks a set of pending events of new data addition request that are candidates for incremental modification of the shared data and adds them to the data when it believes consensus has been reached. Consensus is an algorithm that enables multiple nodes in a network to make a decision in a deterministic fashion.

### Data Model

Incremental modification is implemented as addition of a new portion of formatted data called block. All blocks are organized into a linear sequence over time (also known as the blockchain). In other word new blocks are constantly being processes by nodes and added to the end of the chain. The blocks must not be changed or removed once accepted by the network otherwise the data consistency is violated – the chain is broken.

Because the system is decentralized it is possible that the chain is forked incidentally. In this case the longest chain serves as proof of the sequence of events (addition operations) and the orphaned blocks are removed.

The block is a data structure that contains one or more data entities and metadata used to organize the chain of blocks. The network timestamps the blocks by hashing them into an ongoing chain. Each block contains, among other things, a set of user data entities, and a reference to the block that came immediately before it. It also contains a digest (hash value) used to protect the chain integrity, i.e. using the hash value the ordering of the blocks can be mathematically proven to be correct. Data integrity is the assurance of the accuracy and consistency of the blocks content over its entire life-cycle.

### Node Agent

The node agent is a software program that must handle simultaneously two interfaces:

* Network Attachment Point (NAP) to be exposed to the network managing the chain of blocks;
* Application Program Interface (API) to be exposed to an upper layer client application (proprietary user software) reading and modifying the data stored in the blocks.

A common peer-to-peer protocols suit has to be used to assure interoperability of the node agents exposed by the NAP to the network. Functionality and implementation technology of the API depends on the proprietary requirements of the client application.

To maintain and protect the chain of blocks by executing algorithms according to blockchain rules the node agent is responsible to support the following tightly coupled functionality:

* Clones the current chain of blocks to create a local copy on the agent computer;
* Shares the local copy of the chain of blocks with other nodes;
* Calculates the hash-based digest and formats blocks according to a specific for the network syntax and semantics as the result of the client application request;
* Tracks a set of pending data add requests that are candidates for being attached to the chain and communicates with other nodes to realize consensus mechanism and finally adds the new blocks to the chain;

Exchanging the data over the network the agents validates the block consistency as a one whole during cloning the chain of blokes at startup or incrementally while new blocks are committed to the local repository. It is worth stressing that the incremental validation is applied only to new blocks attached to the end of the chain. Because cloning of the local copy of chain of blocks from the network may be very time expensive operation any request issued by the client application using the agent API should be fulfilled using local data storage. Additionally assuming that the block read request is achieved by performing a procedure call on the block identifier the agent must support random data access. It is impractical to use the digest to confirm data integrity during retrieval of the block. Additionally the agent may maintain only partial copy of the shared chain of blocks, so the procedure may return an error if the indemnifier is not found.

### Domain Model



Figure Node Agent Domain Model

In Figure 1 a domain model of the node agent software is shown. **Node Agent API** exposes the agent functionality to the upper layer client. **Node Agent Services** is responsible to handle the **Client** requests and interoperate with other nodes over the network. The interoperation requires realization of **Consensus** mechanism and removing orphaned blocks using **Fork Protection** functional class. It clones the chain of blocks using **Replication** and maintains local copy of the chain of blocks using the **Storage** as a part of **Infrastructure**.

To expose the chain of blocks to the network the first and last block must be known. Blocks are associated with each other to create stream.

## Architektura BlockGraph

### Wstęp

W rozdziale tym opisano proponowane rozwiązanie dla architektury BlockGraph, które umożliwiają zapisanie grafów RDF w blokach tworzących łańcuch zgodnie z technologią Blockchain. Rozwiązania te zaprojektowano i zaimplementowano tak, aby umożliwić aplikacji użytkowej dostęp do danych użytkowych z wykorzystaniem realizacji kwerend SPARQL. Proponowane API rozwiązania umożliwia również realizację typowych operacji na danych łańcuchowych zgodnie z technologią Blockchain. API rozwiązania rozszerzono o dedykowane operacje pozwalające na dodawanie do łańcucha danych RDF. Rozdział 3.2.2 zawiera szczegółowy opis proponowanych rozwiązań.

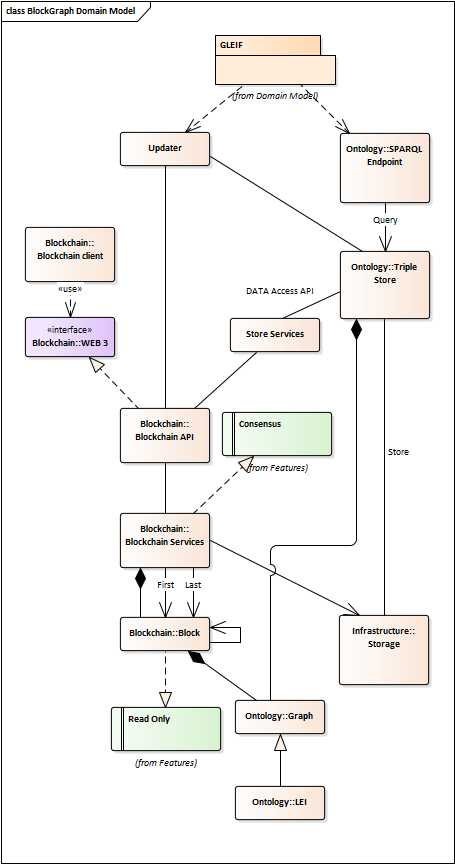
W celu sprawdzenia wykonalności proponowanych rozwiązań zbudowano stanowisko badawcze wyposażone w autorskie oprogramowanie implementujące niezbędne funkcje. Rozwiązanie to wykorzystuje istniejące biblioteki, które zmodyfikowano i zintegrowano w celu otrzymania spójnego rozwiązania. Rozdział 3.2.3 przedstawia szczegółowy opis przeprowadzonych badań przemysłowych w tym zakresie.

### Proponowane rozwiązanie

W architekturze BlockGraph (patrz rysunek poniżej) dane udostępnione są poprzez następujące klasy funkcyjne:

* Blockchain client: typowy interfejs dla oprogramowania węzła implementującego technologię Blockchain;
* Updater: oferuje aplikacji użytkowej możliwość dodawania do łańcucha kolejnych bloków z danymi;
* SPARQL Endpoint: zapewnia możliwość wykonywania kwerend SPARQL na bazie grafowej utworzonej z danych przechowywanych w łańcuchu

Możliwość dostępu do typowych funkcji Blockchain poprzez klasę Blockchain client pozwala na dodawanie do łańcucha danych o dowolnym typie, które mogą być zapisane z wykorzystaniem reguł składniowych i semantycznych bloku w danej implementacji technologii Blockchain.

Klasy Updater i SPARQL Endpoint zapewniają aplikacji użytkowej (np. [lei.info](file:///C:\Users\mariusz.postol\AppData\Roaming\Microsoft\Word\lei.info)) dostęp z wykorzystaniem danych grafowych. SPARQL Endpoint to wybrana implementacja kompatybilna z językiem SPARQL (ang. Protocol And RDF Query Language). Ponieważ Blockchain jest repozytorium, które nie pozwala na operację modyfikowania danych, a w tym ich kasowania oraz dane mogą być tylko dodawane na koniec tworzonego łańcucha w wyniku realizacji operacji uzyskania konsensusu wykonywanie funkcji dodaj, kasuj i modyfikuj dostępnych w typowym interfejsie SPARQL Endpoint muszą być zablokowane by nie naruszyć spójności łańcuch danych. Z uwagi na konieczność uzyskania konsensusu przed dodaniem kolejnych danych do łańcucha operacja ta musi być delegowana do osobnej klasy. W opisywanej architekturze jest nią Updater. Odpowiada ona za realizację żądań zgłaszanych przez aplikację użytkową dodania danych.

Klasa Store Services pełni w omawianej architekturze funkcję synchronizacji danych zarządzanych przez Blockchain Services i Triple Store. W celu permanentnego przechowywania dedykowanych danych lokalnych przez obie klasy są one dodatkowo powiązane klasą Storage.

W tej architekturze klasa Triple Store może być wykorzystany w dwóch rolach:

* Cache – gdy dane są wprowadzane jednorazowo przy inicjacji przez Store Service i modyfikowane przez Updater za każdym razem, kiedy nowe dane zostaną dołączone do łańcucha
* Proxy - gdy dane są wprowadzane każdorazowo przed wykonaniem kwerendy żądanej przez aplikację użytkową.

Ponieważ podstawowym założeniem proponowanych rozwiązań jest przechowywanie danych typu RDF, składnia i semantyka zawartości bloku musi to umożliwiać. Warunek ten można spełnić w wyniku:

* odpowiedniego mapowania danych RDF na dane zgodne z wybranym standardem
* opracowania własnej składni i semantyki bloku

W celu zwiększenia elastyczności i zapewnienia otwartości rozwiązania poprzez umożliwienie zapisu w łańcuchu również danych nie limitowanych przez zgodność z RDF, w proponowanym rozwiązaniu opracowano własną składnię i semantykę bloków. Opis proponowanych rozwiązań znajduje się w dokumencie umieszonym w publicznym repozytorium pod adresem:

<https://github.com/MakoLab/bds-docs/blob/master/GraphBlockContent.md>

Warto tu podkreślić, że proponowane zmiany zapewniając swobodę wyboru rodzaju zapisywanych danych przy jednoczesnym zagwarantowaniu, że poziom zabezpieczenia łańcucha pozostanie bez mian, a mianowicie spójność bloków jest kontrolowana przez zgodność wartość skrótu wyliczonego przy tworzeniu bloku i sprawdzanego przy jego odczycie zgodnie z wzorem:

HC(BlobC) = HR(BlobR)

gdzie:

* HC , HR - funkcje skrótu odpowiednio dla tworzenia bloku i odczytu danych;
* BlobC, BlobR - zawartość bloku odpowiednio dla tworzenia i odczytu danych;

### Implementacja proponowanych rozwiązań

Ponieważ możliwość wykonania kwerendy SPARQL jest jedną z kluczowych funkcjonalności każdego Triplestore’a, niezbędny jest test wykonalności takiego zapytania w implementacji BlockGraph.

Do testów utworzono plik typu NQuad z 500000 grafów. Pojedynczy test polegał na:

* Pobraniu z pliku określonej ilości grafów i utworzeniu z nich w pamięci łańcucha bloków (jeden graf per blok)
* Załadowaniu w.w. grafów do pamięciowej implementacji Triplestore’a (dotNetRdf)
* Wykonaniu kwerendy na utworzonym Triplestorze
* Testy wykonano dla prób 10, 100, 1000, 10000 i 100000 grafowych.

Każdy graf w bazie poddawany jest tym samym operacjom, a wszystkie grafy przetwarzane są sekwencyjnie, co w rezultacie daje złożoność obliczeniową O(N), gdzie N to liczba grafów w bazie.

API rozwiązania jest określone przez wykorzystaną do testów bibliotekę dotNetRDF. Umożliwia ona swobodny dostęp do grafów oraz możliwość wykonywania kwerend SPARQL. W opisanych testach użyto dostępu poprzez kwerendy SPARQL.

Opisywany test miał przede wszystkim za zadanie zbadać wykonalność funkcjonalności zapytań SPARQL dla architektury BlockGraph. Testy wykonywano bez wykorzystania pośredniej bazy Triplestore typu cache.

By umożliwić przechowywanie danych RDF w blokach Blockchain w wykorzystanym do testów oprogramowaniu NStratis dokonano modyfikacji, polegającej na utworzeniu uniwersalnej klasy pozwalająca na przechowywanie dowolnego typu, a tym grafów RDF. W celu utworzenia w/w klasy GraphBlock wyodrębniono z istniejącej klasy bloku wszystkie wspólne metody i właściwości i użyto ich do utworzenia interfejsu i klasy bazowej. Nowopowstała klasa GraphBlock zastąpiła istniejącą w klasie pierwotnej kolekcję transakcji kolekcją obiektów klasy GraphContent, która przetrzymuje informację o zawartości bloku oraz niezbędne informacje nagłówkowe. Zastąpiono również metody służące do dodawania transakcji metodami dodającymi grafy. Wspomniany nagłówek, wzorowany na nagłówku protokołu http, pozwala na opisanie typu danych zapisanych w bloku, a także sposobu serializacji zapisanego grafu. W klasie GraphBlock struktura nagłówka bloku pozostała bez zmian, a także metody wyliczania skrótu nagłówka, zaś obliczanie MerkleRoot dla danego bloku przebiega w sposób analogiczny do obliczania MerkleRoota dla bloku Blockchain, dzięki czemu sposób zabezpieczania łańcucha proponowanego rozwiązania jest identyczny z oryginałem.

Kod programu umieszczono w repozytorium publicznym dostępnym pod adresem:

* <https://github.com/MakoLab/blockgraph>

## Architektura GraphChain

### Wstęp

W rozdziale tym opisano proponowane rozwiązanie dla architektury GraphChain, które umożliwia utworzenie zabezpieczonego kryptograficznie łańcucha grafów RDF zgodnie z technologią Blockchain. Rozwiązania te zaprojektowano i zaimplementowano tak, aby umożliwić:

* aplikacji użytkowej swobodne wykonywanie kwerend SPARQL pozwalających na odczyt danych z grafowej bazy danych
* utrzymywanie wspólnego łańcucha grafów RDF przez węzły połączone w sieć

Metody dostępne poprzez API rozwiązań rozszerzono o dedykowane operacje pozwalające na dodawanie do łańcucha nowych danych RDF. Rozdział poniżej zawiera szczegółowy opis proponowanych rozwiązań.

W celu sprawdzenia wykonalności proponowanych rozwiązań stworzono stanowisko badawcze wyposażone w autorskie oprogramowanie implementujące niezbędne funkcje. Rozwiązanie to wykorzystuje istniejące biblioteki, które zmodyfikowano i zintegrowano w celu otrzymania spójnego rozwiązania zgodnego z opisem w rozdziale 3.3.2. W kolejnych rozdziałach przedstawiono szczegółowy opis przeprowadzonych badań przemysłowych w tym zakresie.

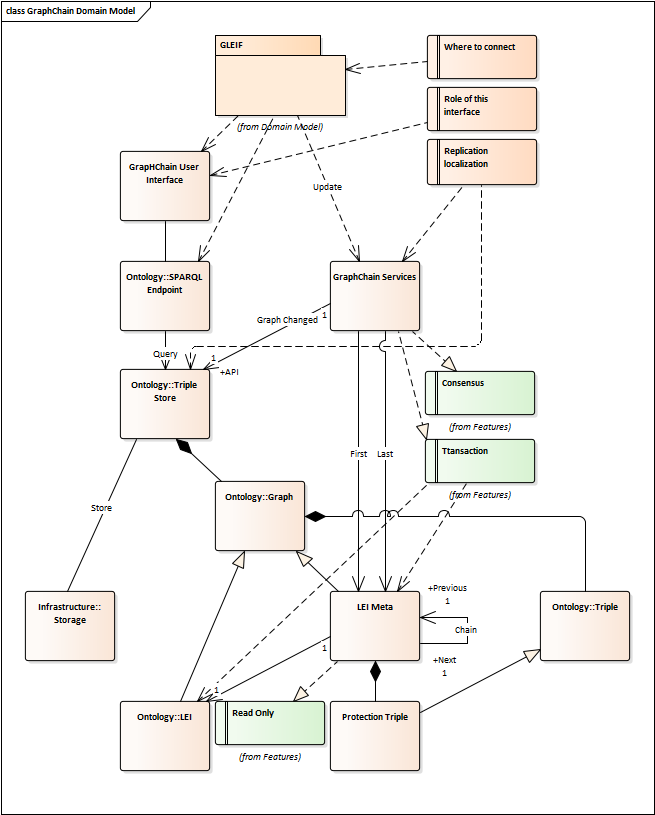
### Proponowane rozwiązania

W architekturze GraphChain (patrz rysunek poniżej) dane udostępnione są aplikacji użytkowej (np. [lei.info](file:///C:\Users\mariusz.postol\AppData\Roaming\Microsoft\Word\lei.info)) poprzez następujące klasy:

* GraphChainServices: oferuje aplikacji użytkowej możliwość dodawania do łańcucha kolejnych bloków z danymi;
* SPARQL Endpoint: zapewnia możliwość wykonywania kwerend SPARQL na bazie grafowej, w której grafy połączono w łańcuch i zabezpieczono metodami kryptograficznymi ekwiwalentnymi do tych stosowanych w technologii Blockchain.

Poprzez klasę GraphChainServices aplikacja użytkowa wysyła żądania dodania nowego grafu RDF do łańcucha. Jest ona również odpowiedzialna za synchronizację replik bazy grafowej na różnych węzłach połączonych w sieć. Klasa ta musi realizować typowe funkcje Blockchain związane z komunikacją pomiędzy węzłami w celu replikacji łańcucha grafów na innych węzłach i uzgadniania z nimi jego wszelkich modyfikacji. Ponieważ w proponowanym rozwiązaniu dane łańcucha są przechowywane w repozytorium typu Triplestore, klasa GraphChainServices musi również odpowiadać za generowanie dodatkowych grafów, które odpowiedzialne są za utworzenie wirtualnego łańcucha grafów reprezentujących dane użytkowe i kryptograficzne zabezpieczenie jego spójności zgodnie z paradygmatem Blockchain.

SPARQL Endpoint zapewnia aplikacji użytkowej (w naszym przypadku GLEIF) dostęp do danych użytkowych z wykorzystaniem kwerend SPARQL. SPARQL Endpoint to wybrana implementacja kompatybilna z językiem SPARQL. Ponieważ zgodnie z paradygmatem technologii Blockchain dane wpisane raz do repozytorium nie mogą być modyfikowane w żadnym przypadku oraz dane mogą być tylko dodawane na koniec tworzonego łańcucha w wyniku realizacji operacji uzyskania konsensusu, wykonywanie funkcje dodaj, kasuj i modyfikuj dostępne w typowym interfejsie SPARQL Endpoint muszą być zablokowane by nie naruszyć spójności łańcuch danych. Z uwagi na konieczność uzyskania konsensusu przed dodaniem kolejnych danych do łańcucha, żądanie dodania musi być delegowane do osobnej klasy. W opisywanej architekturze jest nią GraphChainServices.



Ponieważ podstawowym założeniem proponowanych rozwiązań jest przechowywanie i zabezpieczenie danych użytkowych w postaci grafów RDF, dlatego realizacja architektury GraphChain wymaga opracowania ontologii reprezentującej strukturę łańcucha bloków w postaci łańcucha grafów zgodnie z paradygmatem technologii Blockchain. Opracowana na potrzeby realizacji omawianej architektury ontologia jest opisana w rozdziale 3.3.3.

### Opracowanie koncepcji struktury grafu do zrealizowania zabezpieczeń kryptograficznych typowych dla Blockchain.

Realizacja architektury GraphChain wymagała opracowania ontologii reprezentującej strukturę bloku oraz całego łańcucha grafów tak, jak zostało to przedstawione w pracy S. Nakamoto [S. Nakamoto 2008]. Opracowana ontologia GraphChaina jest dostępna w publicznym repozytorium pod adresem:

<https://github.com/MakoLab/graphchain/blob/master/OntoBC.ttl>.

Podstawowym celem jest możliwość generowanie grafu stanowiącego semantyczny odpowiednik nagłówka pakietu używanego w technologii Blockchain dla grafów RDF zawierających dane (w naszym przypadku były to grafy reprezentujące LEI (ang. Legal Entity Identifier)). Poniższy rysunek przedstawia schemat bloku oraz wszystkie relacje wraz z restrykcjami. Widać, że relacje mają charakter obligatoryjny i są funkcjami.



Poniższy kod RDF (w serializacji Turtle) przedstawia przykładowy blok Genesis oraz kolejny blok. Można zaobserwować, że każdy z tych bloków reprezentuje informacje ekwiwalentne z danymi dla Block Header w typowym rozwiązaniu Blockchain, tj. identyfikator poprzedniego bloku (bc:hasPreviousBlock), IRI grafu z danymi (bc:hasDataGraphIRI), wartość skrót dla grafu z danymi (bc:hasDataHash), wartość skrótu dla całego bloku (bc:hasHash), indeks bloku (bc:hasIndex), wartość skrótu dla poprzedniego bloku (bc:hasPreviousHash) oraz czas utworzenia bloku (bc:hasTimeStamp).

@prefix bc: <http://www.ontologies.makolab.com/bc/> .

@prefix owl: <http://www.w3.org/2002/07/owl#> .

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

@prefix xml: <http://www.w3.org/XML/1998/namespace> .

@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

bc:GenesisBlock a owl:NamedIndividual , bc:GenesisBlock ;

bc:hasPreviousBlock :GenesisBlock ;

bc:hasDataGraphIRI "http://ontologies.makolab.com/bc"^^xsd:anyURI ;

bc:hasDataHash "1234714f85f6f443930cf2b5c6c6323b2ccc1f111cd52a16df1dcbc2735e5678"^^xsd:string ;

bc:hasHash "9123714f85f6f443930cf2b5c6c6323b2ccc1f111cd52a16df1dcbc2735easdf"^^xsd:string ;

bc:hasIndex "0"^^xsd:decimal ;

bc:hasPreviousHash "0"^^xsd:string ;

bc:hasTimeStamp "1502269780"^^xsd:decimal .

bc:Block1 a owl:NamedIndividual , bc:Block ;

bc:hasPreviousBlock :GenesisBlock ;

bc:hasDataGraphIRI "http://lei.info/6SHGI4ZSSLCXXQSBB395"^^xsd:anyURI ;

bc:hasDataHash "84fa714f85f6f443930cf2b5c6c6323b2ccc1f111cd52a16df1dcbc2735eb9e2"^^xsd:string ;

bc:hasHash "543a714f85f6f443930cf2b5c6c6323b2ccc1f111cd52a16df1dcbc2735eb456"^^xsd:string ;

bc:hasIndex "1"^^xsd:decimal ;

bc:hasPreviousHash "9123714f85f6f443930cf2b5c6c6323b2ccc1f111cd52a16df1dcbc2735easdf"^^xsd:string ;

bc:hasTimeStamp "1502267930"^^xsd:decimal .

### Stworzenie funkcjonujących mechanizmów kryptograficznych

#### Wstęp

Graf RDF może być zapisany (serializowany) z wykorzystaniem różnych reguł składniowych. Dodatkowo w ramach jednego standardu semantycznie ten sam graf może być zapisany z wykorzystaniem różnych, ale poprawnych składniowo ciągów znaków. W związku z tym reprezentacja grafu musi być znormalizowana tak, by wyliczanie wartości kryptograficznej funkcji skrótu (dalej zwanej skrótem) dawało zawsze taki sam wynik dla tego samego grafu. Na podstawie analizy literatury i zrealizowanych prac badawczych wybrano metody, które opisano w kolejnych rozdziałach. Ich implementację zrealizowano w oprogramowaniu potwierdzającym możliwość praktycznego wdrożenia architektury GraphChain. Kod programu jest dostępny pod adresem:

<https://github.com/MakoLab/graphchain>

Wyboru metody CircleDot Hashing dokonano biorąc pod uwagę jej istotną cechę, a mianowicie pozwala ona na niezwykle szybkie wyliczenie wartości funkcji skrótu całego grafu wtedy, gdy zachodzi w nim modyfikacja istniejących lub dodanie nowych elementów. W tym wypadku skrót całego grafu może być obliczony analitycznie z numerycznym obliczaniem tylko zmienianej (lub nowej) sekwencji trójek grafu. Ta cecha tej metody może mieć potencjalnie duże znaczenie w przyszłych zastosowaniach GraphChaina i zostanie dokładniej zbadana w kolejnych etapach projektu.

Ponadto ważną cechą tej metody jest możliwość relatywnie prostej w realizacji kodu obliczania takiej funkcji skrótu z wykorzystaniem programowania równoległego, co pozwala na N-krotne skrócenie czasu obliczeń skrótu na popularnych procesorach „z N cores.” Możliwość ta zostanie również zbadana dokładniej w kolejnych etapach projektu.

#### Metoda CircleDot Hashing

Wyliczenia skrótu grafu z danymi wykonywane jest z wykorzystaniem metody opracowanej na podstawie propozycji przedstawionej w pracy Sayers'a i Karp'a [Sayers, Karp 2004]"i nazwanej CircleDot Hashing. Ten sposób wyliczania wartości funkcji skrótu można opisać następującym wzorem:

D(S) jest algorytmem przyjmującym na wejściu zbiór trójek *S* z danego grafu i zwracającym skrót dla tego grafu zaś *CircleDot* jest funkcją przyrostowo wyliczającą skrót dla zbioru *N* trójek. Do obliczenia kolejnych iteracji używane jest dodawanie z operacją modulo. *SHA-256* jest nazwą implementacji wykorzystywanego algorytmu kryptograficznego wyliczania skrótu, zaś *NTriplesSerialization* nazwą funkcji serializującej poszczególne trójki za pomocą standardu N-Triples.

Korzystając z kryptograficznej funkcji skrótu po wyliczeniu jej wartości dla grafu (dalej nazywanej skrótem), jak to opisano wyżej, następują kolejno: stworzenie obiektu nagłówka bloku, stworzenie obiektu treści bloku oraz przetworzenie obiektu nagłówka bloku na reprezentację trójkową.

Po ukończeniu procedury opisanej w zadaniu „CircleDot Hashing Block Creation” następuje zapisanie danych w RDF4J Server, uruchomionym jako usługa na tej samej maszynie.

Walidacja łańcucha zabezpieczonego poprzez CircleDot Hashing wykonywana jest w następujących krokach: odczytanie ostatniego bloku (nagłówek i dane) z repozytorium, wyliczenie funkcji skrótu z użyciem algorytmu CircleDot, opisanego powyżej, sprawdzenie, czy wyliczone skrótu są takie same jak wartości w repozytorium. Jeżeli skrótyy nie są takie same, zwracana jest informacja o niepoprawności bloku. W przeciwnym przypadku, następuje sprawdzenie czy blok nie jest pierwszym blokiem w łańcuchu (tzw. *genesis block*). Jeżeli tak, to program kończy swoje działanie. W przeciwnym przypadku, odczytany zostaje poprzedni blok.

#### Metoda JsonLd Hashing

Wyliczenie skrótu dla całego grafu z danymi zakłada jego wcześniejszą normalizację. Normalizacja grafu dokonuje się z wykorzystaniem procedury normalizacyjnej (JsonLdProcessor.normalize) dostępnej w bibliotece [JsonLd](https://json-ld.org/) dla Javy. Wynikiem działania tej procedury jest obiekt typu String ze znormalizowaną reprezentacją grafu w formacie JsonLd. Stworzenie skrótu dla znormalizowanego grafu dokonuje się z wykorzystaniem funkcji SHA-256. Przed wywołaniem tej metody dokonywana jest konwersja obiektu String na tablicę bajtów z użyciem kodowania UTF-8.

Po wyliczeniu skrótu grafu następują kolejno kroki: stworzenie obiektu nagłówka bloku, stworzenie obiektu treści bloku oraz przetworzenie obiektu nagłówka bloku na reprezentacje trójkową. Po ukończeniu tej procedury następuje zapisanie danych w RDF4J Server.

Walidacja łańcucha wykonana jest w następujących krokach: odczytanie ostatniego bloku (nagłówek i dane) z repozytorium, przeprowadzenie procesu normalizacji z wykorzystaniem algorytmu normalizującego z biblioteki JsonLd dla Javy, ponowne wyliczenie skrótu dla danych znormalizowanych oraz skrótu dla bloku, sprawdzenie, czy wyliczone skróty są takie same jak skróty w repozytorium. Jeżeli skróty nie są takie same, zwracana jest informacja o niepoprawności bloku. W przeciwnym przypadku, następuje sprawdzenie czy blok nie jest pierwszym blokiem w łańcuchu (tzw. *genesis block*). Jeżeli tak, to program kończy swoje działanie. W przeciwnym przypadku, odczytany zostaje poprzedni blok.

### Stworzenie mechanizmów replikacji struktur grafowych

Jako punkt odniesienia do przygotowania aplikacji typu PoC (ang. Proof of Concept) implementującej mechanizmy replikacji struktur grafowych przyjęto [Naivechain](https://github.com/lhartikk/naivechain). Realizuje ona prosty mechanizm typu p2p (ang. peer-to-peer). Do dalszych prac proponowane były m.in.:

* natywne mechanizmy replikacji używane w triplestore'ach, np. [AllegroGraph](https://franz.com/agraph/support/documentation/current/replication.html)
* implementacje protokołów p2p obecne w platformach blockchain, np. [ethereumjs-devp2p](https://github.com/ethereumjs/ethereumjs-devp2p)

Należy jednak pamiętać, że natywne mechanizmy replikacji triplestore'ów nie muszą mieć żadnych mechanizmów zabezpieczeń obecnych w blockchain'ach, mogą więc wymagać niezbędnej modyfikacji.

### Stworzenie warstwy dostępowej (HTTP API) dla tak przygotowanej implementacji bazy typu RDF Graph.

HTTP API aplikacji realizującej architekturę GraphChain zostało udokumentowane w plikach: APIblock.md, APIblock\_.md, APIblock\_create.md umieszczonych w repozytorium publicznym:

<https://github.com/MakoLab/graphchain/tree/master/GraphChainAPI>

### Implementacja proponowanych rozwiązań

Testowanie praktyczne architektury GraphChain miało na celu sprawdzenie wydajności tworzenia łańcucha bloków. Na poszczególne bloki składa się zarówno treść grafu wejściowego (w tym teście używano grafów z danymi LEI) jak i graf zawierający metadane chroniące spójność łańcucha grafów. Zbadaniem wydajności procesu normalizacji grafu było konieczne ze względu na powtarzalność operacji tworzenia kryptograficznej funkcji skrótu dla danego grafu. Przetestowano wydajność dwóch sposobów wyliczania skrótu dla grafu: (1) z wykorzystaniem metody normalizacji z biblioteki JsonLd dla Javy; (2) z wykorzystaniem autorskiej implementacji inkrementacyjnego wyliczania skrótu opartego na algorytmie opisanym w części 3.3.4.2 raportu. Jeden z testów uwzględnia również czas potrzebny na zapisanie grafów i stworzonych dla nich bloków w lokalnym (tj. działającym jako usługa na komputerze, na którym były wykonywane testy) repozytorium trójek (tzw. *triple store*). Zbadano również wydajność programu walidującego graf dla obu powyższych rozwiązań normalizacyjnych. W testach, w których wymagane jest pobieranie danych z repozytorium trójek, używane są kwerendy SPARQL. Jest to standardowy sposób komunikacji z repozytoriami trójek, który – przynajmniej do pewnego stopnia – pozwala uniknąć zależności od specyficznych produktów. Testy zostały przeprowadzone na komputerze o następujących parametrach: Intel Core i7-3610QM CPU @ 2.30 GHz x 8, 16 GB RAM, 256 GB SSD, Ubuntu 16.10 64-bit. W testach, w których wykorzystano repozytorium trójek, użyto RDF4J Server w wersji 2.2.2. W zadaniu „JsonLd Hashing” wykorzystano bibliotekę JsonLd dla języka programowania Java, która w repozytorium Mavena jest dostępna jako: groupId=com.github.jsonld-java, artifactId=jsonld-java, version=0.8.3.

Poniższe tabele zawierają wyniki testów wydajności dla metod opisanych w rozdziałach 3.3.4.2 i 3.3.4.3. Czasy zapisane są w sekundach. Kolumna z nagłówkiem „Liczba LEI” zawiera liczbę grafów (do testów zostały użyte rzeczywiste grafy z danymi LEI), na których przeprowadzony został test. Na podstawie przedstawionych wyników można stwierdzić, że czas potrzebny na proces normalizacji grafu wejściowego oraz utworzenie na jego podstawie bloków jest liniowy, tzn. zwiększa się proporcjonalnie wraz ze wzrostem liczby wejściowych grafów.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Liczba LEI | CircleDot Hashing (s) | CircleDot Hashing Block Creation (s) | JsonLd Hashing (s) | JsonLd Hashing Block Creation (s) |
| 10 | 0.089 | 0.08 | 0.228 | 0.225 |
| 100 | 0.237 | 0.298 | 0.486 | 0.622 |
| 1000 | 1.53 | 1.677 | 2.649 | 3.003 |
| 10000 | 13.085 | 15.345 | 22.441 | 25.886 |
| 100000 | 132.916 | 153.412 | 226.347 | 253.88 |

Na podstawie testu można stwierdzić, że implementacje wykorzystujące algorytm CircleDot są mniej kosztowne obliczeniowo. W poniższej tabeli widać wyraźne odejście od wzrostu liniowego, zwłaszcza dla 10000 grafów, co prawdopodobnie wiąże się ze zmniejszeniem się wydajności działania używanego w testach repozytorium trójek wraz ze wzrostem liczby zapisanych w nim trójek. Zaprezentowany w tabeli poniżej wzrost ilości czasu potrzebnego do walidacji określonej liczby bloków nie jest wzrostem liniowym. Fakt ten można próbować wyjaśnić tym, że na skutek przechowywania większej liczby trójek w repozytorium, czas potrzebny na wykonanie zapytań SPARQL-owych do tego repozytorium jest coraz większy. Przy większej liczbie grafów różnica pomiędzy szybkością wykonania obu algorytmów przestaje być zauważalna.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Liczba LEI | CircleDot Hashin Block Creation (s) | JsonLd Hashing Block Creation (s) | JsonLd BlockChain Validation (s) | CircleDot BlockChain Validation (s) |
| 10 | 1.1 | 1.6 | 0.3 | 0.1 |
| 100 | 9.8 | 11.4 | 1.8 | 0.9 |
| 1000 | 114.0 | 122.2 | 26.2 | 24.6 |
| 10000 | 3287.1 | 3586.8 | 2007.3 | 2052.9 |

Kody oprogramowania umieszczono w repozytorium:

* <https://github.com/MakoLab/graphchain>

# Badanie implementacji proponowanych rozwiązań

## Zakres badań i wybór kryteriów oceny

W celu potwierdzenie wykonalności zaproponowanych rozwiązań zaimplementowano wybrane bloki funkcjonalne dla rozwiązań zaproponowanych w rozdziale 3.

Zrezygnowano z badań scenariusza cache dla BlockGraph – uzasadnić.

## Badanie architektury BlockGraph

Badana architektura została opisana w rozdziale 3.2. Ponieważ możliwość wykonania kwerendy SPARQL jest jedną z kluczowych funkcjonalności każdego triplestore’a, niezbędny jest test wykonalności takiego zapytania w implementacji BlockGraph.

### Opis testu

Do testów stworzyliśmy plik NQuad z 500000 grafów. Pojedynczy test polegał na:

* Pobraniu z pliku określonej ilości grafów i utworzeniu z nich w pamięci łańcucha bloków (jeden graf per blok)
* Załadowaniu ww. grafów do pamięciowej implementacji triplestore’a (dotNetRdf)
* Wykonaniu kwerendy na utworzonym triplestorze.

Testy wykonaliśmy dla prób 10, 100, 1000, 10000 I 100000 grafowych.

Maszyna, na której wykonano testy:

* i7-3610QM 2,3GHz
* 16GB RAM
* 256GB SSD
* Windows 10 Pro x64

### Złożoność obliczeniowa

Wyniki zaprezentowane są na wykresie poniżej. Jak łatwo zaobserwować, czasy tworzenia blockchaina i ładowania triplestore’a są praktycznie liniowo zależne od liczby grafów. (Oś pionowa jest wyskalowana logarytmicznie, ale należy zwrócić też uwagę, że liczba grafów w kolejnych próbach rośnie wykładniczo) Niewielkie odchylenia od tej reguły można zaobserwować przy najmniejszych próbach (wpływ stałych, jednorazowych operacji) oraz przy największej (wpływ wyczerpującej się dostępnej pamięci RAM). Aby dokładniej unaocznić zależność między czasami tworzenia struktur BC i TS posłużymy się drugim wykresem prezentującym czasy tych samych operacji ale w ujęciu per graf. Najbardziej rzuca się w oczy praktycznie identyczny czas testu dla prób 1000 i 10000 co wskazuje na liniową złożoność tych operacji.

Analiza algorytmu testującego potwierdza tę obserwację. Każdy graf w bazie poddawany jest tym samym operacją, a wszystkie grafy przetwarzane są sekwencyjnie co w rezultacie daje złożoność obliczeniową O(N), gdzie N to liczba grafów w bazie.

### Wykonanie kwerendy

Szary pasek na obu wykresach wskazuje czas oczekiwania na wykonanie kwerendy. W większości przypadków liczba grafów nie miała wpływu na ten czas. Wyjątkowo duży czasy wykonania dla ostatniego testu jest najprawdopodobniej spowodowany zbliżeniem się do limitu dostępnej pamięci na maszynie testowej.

Chociaż wynik 327ms dla 10000 grafów jest wynikiem akceptowalnym, tym bardziej że w środowisku produkcyjnym pamięciowa implementacja byłaby zastąpiona dedykowanym serwerem bazodanowym, to należy mieć na uwadze fakt, że w naszym rozwiązaniu nie przewidujemy użycia cache’a, a stały triplestore utworzony na podstawie danych z blockchaina pełniłby właśnie taką funkcję. Aby tego uniknąć, proces tworzenia pamięciowego triplestore’a musiałby być powtórzony dla każdego zapytania, a zatem czas na odpowiedź byłby równy sumie czasów tworzenia TS (pomarańczowy pasek) i wykonania kwerendy (szary pasek). Dla 10000 grafów byłoby to 32384ms.

### Podsumowanie

Przedstawiony test miał przede wszystkim za zadanie zbadać wydajność wykonywania zapytań SPARQL dla testowej implementacji architektury BlockGraph. Założyliśmy w nim brak stałej pośredniej bazy triplestore typu cache. Test wykazał, że o ile sama kwerenda wykonuje się szybko (w większości przypadków, znacznie poniżej sekundy), to potrzeba utworzenia triplestore’a przy każdym zapytaniu wydłuża ten czas do nieakceptowalnych poziomów (kilkadziesiąt sekund). To praktycznie wyklucza działanie endpointu SPARQL w trybie proxy w rozwiązaniu blockgraph.

## Badanie GraphChain

### Wprowadzenie

W celu zbadania praktycznej możliwości implementacji architektury opisane w rozdziale 3.3. Badanie miało na celu sprawdzenie wydajności zadania utworzenia łańcucha bloków. Na poszczególne bloki składa się zarówno treść grafu wejściowego (w tym teście używano grafów z danymi LEI) jak i treści nagłówka utworzonego dla tego grafu. Właściciel produktu wyraził szczególne zainteresowanie zbadaniem wydajności procesu normalizacji grafu, którego przeprowadzenie jest konieczne ze względu na powtarzalność operacji tworzenia hasza dla danego grafu. Przetestowano wydajność dwóch sposobów wyliczania hasza dla grafu:

* z wykorzystaniem metody normalizacji z biblioteki JsonLd dla Javy;
* z wykorzystaniem autorskiej implementacji inkrementacyjnego wyliczania hasza opartego na rozwiązaniu opisanych w (Sayers, Karp 2004)[[1]](#footnote-2).

Oba sposoby zostały dokładniej opisane w dalszej części raportu.

Jeden z testów uwzględnia również czas potrzebny na zapisanie grafów i stworzonych dla nich bloków w lokalnym (tj. działającym jako usługa na komputerze, na którym były wykonywane testy) repozytorium trójek (tzw. *triple store*).

Zbadano również wydajność programu walidującego graf dla obu powyższych rozwiązań normalizacyjnych.

W testach, w których wymagane jest pobieranie danych z repozytorium trójek, używane są zapytania SPARQL, ponieważ jest to standardowy sposób komunikacji z repozytoriami trójek, który – przynajmniej do pewnego stopnia – pozwala uniknąć związania się z jednym produktem.

### Sprzęt i oprogramowanie

Testy zostały przeprowadzone na komputerze o następujących parametrach:

* Intel Core i7-3610QM CPU @ 2.30 GHz x 8
* 16 GB RAM
* 256 GB SSD
* Ubuntu 16.10 64-bit

W testach, w których wykorzystano repozytorium trójek, użyto RDF4J Server w wersji 2.2.2.

W zadaniu „JsonLd Hashing” wykorzystano bibliotekę JsonLd dla języka programowania Java, która w repozytorium Mavena jest dostępna jako: groupId=com.github.jsonld-java, artifactId=jsonld-java, version=0.8.3.

### Normalizacja grafu, tworzenie bloku

#### Opis testowanych zadań

Poniżej znajdują się opisy zadań (i kroków składających się na nie), które podlegały testowaniu wydajności.

#### CircleDot Hashing

Wyliczenia hasza grafu z wykorzystaniem propozycji przedstawionej w Craig Sayers, Alan H. Karp, „Computing the digest of an RDF graph”, HP Laboratories Palo Alto, 2004. Ten sposób wyliczania hasza można opisac następującym wzorem:

,

gdzie:

* D(S) – algorytm przyjmujący na wejściu zbiór trójek S i zwracający hasz dla tego grafu.
* CircleDot – funkcja przyrostowo wyliczająca hasz dla zbioru N trójek. Do obliczenia kolejnych iteracji używane jest dodawanie z operacją modulo.
* SHA-256 – nazwa implementacji wykorzystywanego algorytmu haszującego.
* NTriplesSerialization – nazwa funkcji serializującej poszczególne trójki za pomocą standardu N-Triples.

#### CircleDot Hashing Block Creation

Kroki z zadania „CircleDot Hashing”.

Stworzenie obiektu nagłówka bloku.

Stworzenie obiektu treści bloku.

Przetworzenie obiektu nagłówka bloku na reprezentacje trójkową.

#### JsonLd Hashing

Normalizacja grafu z wykorzystaniem metody normalizacyjnej (JsonLdProcessor.normalize) dostępnej w bibliotece JsonLd dla Javy. Wynikiem działania tej metody jest obiekt String ze znormalizowaną serializacją grafu w formacie JsonLd.

Stworzenie hasza dla znormalizowanego grafu z wykorzystaniem funkcji haszującej SHA-256. Przed wywołaniem metody haszującej dokonywana jest konwersja obiektu String na tablicę bajtów z użyciem kodowania UTF-8.

#### JsonLd Hashing Block Creation

Kroki z zadania „JsonLd Hashing”.

Stworzenie obiektu nagłówka bloku.

Stworzenie obiektu treści bloku.

Przetworzenie obiektu nagłówka bloku na reprezentacje trójkową.

#### Wynik testów wydajnościowych

Poniższa tabela zawiera wyniki testów wydajności dla dwóch powyższych zadań. Czasy zapisane są w sekundach. Kolumna z nagłówkiem „Liczba LEI” zawiera liczbę grafów (do testów zostały użyte rzeczywiste grafy z danymi LEI), na których przeprowadzony został test wydajności.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Liczba LEI | CircleDot Hashing (s) | CircleDot Hashing Block Creation (s) | JsonLd Hashing (s) | JsonLd Hashing Block Creation (s) |
| 10 | 0.089 | 0.08 | 0.228 | 0.225 |
| 100 | 0.237 | 0.298 | 0.486 | 0.622 |
| 1000 | 1.53 | 1.677 | 2.649 | 3.003 |
| 10000 | 13.085 | 15.345 | 22.441 | 25.886 |
| 100000 | 132.916 | 153.412 | 226.347 | 253.88 |

Dane z powyższej tabeli zostały zaprezentowane w formie graficznej na poniższych diagramach, z których w drugim skala w obu osiach została zlogarytmizowana.

### Interpretacja

Na podstawie drugiego z powyższych diagramów widać, że czas potrzebny na proces normalizacji grafu wejściowego oraz utworzenie na jego podstawie bloków jest liniowy, tzn. zwiększa się proporcjonalnie wraz ze wzrostem liczby wejściowych grafów.

Na podstawie testu wydajności można stwierdzić, że implementacje wykorzystujące haszowanie z użyciem algorytmu CircleDot są wydajniejsze.

### Zapisywanie trójek w repozytorium

#### Opis testowanych zadań

Poniżej znajdują się opisy zadań (i kroków składających się na nie), które podlegały testowaniu wydajności.

#### JsonLd Hashing Block Creation With Persisting

Kroki z zadania „JsonLd Hashing Block Creation”.

Zapisanie danych w RDF4J Server uruchomionym jako usługa na tej samej maszynie.

#### CircleDot Hashing Block Creation With Persisting

Kroki z zadania „CircleDot Hashing Block Creation”.

Zapisanie danych w RDF4J Server uruchomionym jako usługa na tej samej maszynie.

#### Wynik testów wydajnościowych

Poniższa tabela zawiera wyniki testów wydajności dla dwóch powyższych zadań. Czasy zapisane są w sekundach. Kolumna z nagłówkiem „Liczba LEI” zawiera liczbę grafów (do testów zostały użyte rzeczywiste grafy z danymi LEI), na których przeprowadzony został test wydajności.

|  |  |  |
| --- | --- | --- |
| Liczba LEI | CircleDot Hashing Block Creation With Persisting (s) | JsonLd Hashing Block Creation With Persisting (s) |
| 10 | 1.122 | 1.596 |
| 100 | 9.826 | 11.409 |
| 1000 | 114.055 | 122.221 |
| 10000 | 3287.144 | 3586.835 |

Dane z powyższej tabeli zostały zaprezentowane w formie graficznej na poniższych diagramach, z których w drugim skala w obu osiach została zlogarytmizowana.

### Interpretacja

W obu zadaniach przedstawionych na powyższych diagramach widać wyraźne odejście od wzrostu liniowego, zwłaszcza dla 10000 grafów, co prawdopodobnie wiąże się ze zmniejszeniem się wydajności działania używanego w testach repozytorium trójek wraz ze wzrostem liczby zapisanych w nim trójek. Implementacja zadania wykorzystująca normalizację z użyciem CircleDot jest wydajniejsza.

### Walidacja łańcucha bloków

#### Opis testowanych zadań

#### JsonLd BlockChain Validation

Odczytanie ostatniego bloku (nagłówek i dane) z repozytorium.

Przeprowadzenie procesu normalizacji z wykorzystaniem algorytmu normalizującego z biblioteki JsonLd dla Javy.

Następuje ponowne wyliczenie hasza dla danych znormalizowanych w poprzednim punkcie oraz hasza dla bloku.

Sprawdzenie, czy wyliczone hasze są takie same jak hasze w repozytorium.

Jeżeli hasze nie są takie same, zwracana jest informacja o niepoprawności bloku. W przeciwnym przypadku, następuje sprawdzenie, czy blok nie jest pierwszym blokiem w łańcuchu (tzw. genesis block). Jeżeli tak, to program kończy swoje działanie. W przeciwnym przypadku, odczytany zostaje poprzedni blok i następuje przejście do punktu 2.

#### CircleDot BlockChain Validation

Odczytanie ostatniego bloku (nagłówek i dane) z repozytorium.

Wyliczenie hasza z użyciem algorytmu CircleDot, opisanego powyżej.

Sprawdzenie, czy wyliczone hasze są takie same jak hasze w repozytorium.

Jeżeli hasze nie są takie same, zwracana jest informacja o niepoprawności bloku. W przeciwnym przypadku, następuje sprawdzenie, czy blok nie jest pierwszym blokiem w łańcuchu (tzw. genesis block). Jeżeli tak, to program kończy swoje działanie. W przeciwnym przypadku, odczytany zostaje poprzedni blok. Przejście do punktu 2.

#### Wynik testów wydajnościowych

Poniższa tabela zawiera wyniki testów wydajności dla zadań „JsonLd BlockChain Validation” i „CircleDot BlockChain Validation”. Czasy zapisane są w sekundach. Kolumna z nagłówkiem „Liczba LEI” zawiera liczbę grafów (do testów zostały użyte rzeczywiste grafy dla LEI), dla których przeprowadzony był test.

|  |  |  |
| --- | --- | --- |
| Liczba LEI | JsonLd BlockChain Validation (s) | CircleDot BlockChain Validation (s) |
| 10 | 0.258 | 0.124 |
| 100 | 1.79 | 0.931 |
| 1000 | 26.178 | 24.584 |
| 10000 | 2007.338 | 2052.932 |

Dane z powyższej tabeli zostały zaprezentowane w formie graficznej na poniższych diagramach, z których w drugim skala w obu osiach została zlogarytmizowana.

#### Interpretacja

Zaprezentowany na wykresie wzrost ilości czasu potrzebnego do walidacji określonej liczby bloków nie jest wzrostem liniowym. Fakt ten można próbować wyjaśnić tym, ze na skutek przechowywania większej liczby trójek w repozytorium, czas potrzebny na wykonanie zapytań SPARQL-owych do tego repozytorium jest coraz większy. Przy większej liczbie grafów różnica pomiędzy szybkością wykonania obu algorytmów przestaje być zauważalna.

# Podsumowanie wyników

Dół formularza

## Złożoność obliczeniowa

Oznaczenia:

* NA - nie ma zastosowana (ang. not applicable)
* N - liczba grafów (ang. number of RDF graph)
* n- liczba trójek w grafie (ang. number of triples in graph)

Scenariusze:

Analizowana dwa scenariusze dla BlockGraph:

* **proxy** - TS pełni rolę proxy, co oznacza konieczność odtworzenia jego zawartości na potrzeby każdej kwerendy
* **cache** - TS pełni rolę proxy, tzn. po odtworzeniu TS wprowadzane są do niego tylko modyfikacje łańcucha

### Tworzenie repozytorium

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | BlockGraph  proxy | BlockGraph  cache | GraphChain |
| Get a serialized RDF named graphs | O(N) | O(N) | O(N) |
| RDF canonicalizing[[2]](#footnote-3) | NA[[3]](#footnote-4) | O(N nlog(n)) | O(N nlog(n)) |
| Add the graph to the chain - calculate metadata | O(N) | O(N) | O(N) |
| Add the graph to repository | O( N) ) | O(N) | O( 2N )[[4]](#footnote-5) |
| TOTAL | O(N) | O(N nlog(n)) | O(N nlog(n)) |

### Odczyt danych z repozytorium

|  |  |  |  |
| --- | --- | --- | --- |
| Operacja | BlockGraph  proxy | BlockGraph  cache | GraphChain |
| Odtworzenie zawartości bazy grafowej | O(N) | NA | NA |
| Zapytanie SPARQL | O(1) | O(1) | O(1) |
| TOTAL | O(N) | O(1) | O(1) |

## Realizacja zabezpieczenia strumienia

### BlockGrapH

***Implementacja***: strumień bloków (rekordów) zawierających ciągi bajtów (Blob)

Kontrola zgodności :

HC(BlobC) = HT(BlobT)

Gdzie:

* HC , HT - funkcje skrótu odpowiednio dla tworzenia bloku i odczytu danych
* BlobC , BlobT - zawartość bloku odpowiednio dla tworzenia i odczytu danych

Warunkiem spójności jest identyczność funkcji skrótu i zawartości bloku.

### GraphChain

HC(EncodingC(CanonicalizationC( ReadC(GraphC) ) ) )= HT(EncodingT(CanonicalizationT(ReadC(GraphT) ) ) )

| Funkcja | Nazwa | Wejście | Wyjście |
| --- | --- | --- | --- |
| H | Liczenie skrótu np. HASH256 | Blob | 256 bitów |
| Encoding | Konwersja na blob | string | Blob |
| Canonicalization | Przekształcenie grafu do postaci znormalizowanej zapełniającej zgodność znakową | string | string |
| Read | Odczyt wybranego grafu z TS | Graph | string |

gdzie:

* Blob - ciąg bajtów
* Graph - wewnętrzna reprezentacja grafu zależna od użytego oprogramowania TS

Warunkiem spójności jest identyczność realizacji następujących funkcji:

* Encoding
* Canonicalization( Read( Graph )

Uwaga:  W przypadku GraphChain wartość Graph ma tylko pośrednie znaczenie. Innymi słowy można sobie wyobrazić funkcję Canonicalization( Read( Graph )) zwracającą niezależnie od Graph tą samą wartość by zapewnić zgodność. To ma bezpośredni wpływ na prawdopodobieństwo powstania kolizji funkcji skrótu, więc trzeba udowodnić, zastosowane metody normalizacji nie degradują bezpieczeństwa.

* Podwyższony stopień bezpieczeństwa oraz oczekiwana szybkość dostępu zostały zdefiniowane w sekcji 3 niniejszego wniosku (skopiowałem do opisu powyżej). UWAGA! W sekcji 3 mówi się jeszcze o **niepodważalnej identyfikacji**:
* Niepodważalna identyfikacja w zdecentralizowanym systemie - Technologia Blockchain zastosowana w produkcie wnioskodawcy zapewnia, iż nie istnieje praktyczna możliwość dokonania zmiany lub naruszenia integralności raz zapisanych danych w bazach produktu.
* Produkt będzie oferował najwyższy możliwy poziom niepodważalności danych identyfikacyjnych – uzyskując tym samym największe zaufanie rynku.
* Baza danych produktu oparta o technologię Blockchain będzie funkcjonować w modelu rozproszonym z początkową ilością węzłów powyżej 200 węzłów zaufanych (aktywnych - z prawem zapisu) i ponad 1000 dla węzłów  pasywnych (tylko z prawem odczytu).
* W wypadku braku pozytywnego wyniku, rozumianego jako spełnienie punktów (a) i (b) wnioskodawca będzie mógł kontynuować prace w oparciu o obecne rozwiązania reprezentacji danych.
* Należy tu zaznaczyć, że w wypadku niepowodzenia realizacji tego etapu, istotne dla kontynuacji projektu jest skuteczne zakończenie Etapu 2. Należy to rozumieć w ten sposób, iż nie możliwe jest kontynowanie projektu, jeśli oba Etapy badań przemysłowych (1) i (2) zakończyłyby się niepowodzeniem.

# Wnioski końcowe

## Wstęp

Efektem końcowym etapu jest rozstrzygnięcie, która z proponowanych architektur może być polecona do dalszych badań biorąc pod uwagę takie kryteria jak: bezpieczeństwa, wydajności oraz elastyczności. Dodatkowo należy podkreślić, że aktualnie zaproponowana architektura BlockGraph wymaga przechowywania tych samych danych w dwóch miejscach, co z uwagi na możliwość utraty synchronizacji obu kopii nie jest rekomendowane z punktu widzenia dobrych praktyk inżynierii oprogramowania.

## Bezpieczeństwo danych

Z punktu widzenia bezpieczeństwa, na podstawie analizy cech przedstawionych rozwiązań architektonicznych, można wydzielić następujące całkowicie niezależne obszary, w których dane są dostępne:

* wymiana danych pomiędzy węzłami
* przechowywanie lokalnej kopii danych (łańcucha bloków)
* wymiana danych pomiędzy oprogramowaniem węzła i aplikacją użytkową.

Poziom bezpieczeństwa danych wymienianych pomiędzy węzłami zależy wyłącznie od implementacji protokołu komunikacyjnego i zaimplementowanych mechanizmów uzgadniania. Podstawą bezpieczeństwa jest tu pozytywna akceptacja żądania przyłączenia nowego bloku lub klonowania istniejącego bloku do tworzonego lokalnie łańcucha, która musi być zaakceptowana przez większość węzłów. W tej operacji sprawdzane są dwa warunki decydujące o jego poprawności:

* spójność jego zawartości z ostatnim blokiem w łańcuchu
* brak rozwidlenia łańcucha.

Innymi słowy na bezpieczeństwo danych w obszarze wymiany pomiędzy węzłami proponowane rozwiązania architektoniczne nie mają żadnego wpływu.

Sposób zabezpieczenia lokalnej kopii danych nie jest uzależniony od zastosowania technologii Blockchain. Zasadniczym celem przechowywania nagłówków bloków w kopii lokalnej danych jest możliwość ich wykorzystania przy replikacji bloku do innych węzłów za pośrednictwem sieci komunikacyjnej. Możliwe są jeszcze dwa inne scenariusze wykorzystania danych nagłówkowych lokalnej kopii do kontroli spójności:

* sprawdzanie na potrzeby replikacji przy każdym żądaniu nadchodzącym z sieci
* sprawdzenie na potrzeby aplikacji użytkowej.

W przypadku, gdy żądania replikacji kolejnych bloków przychodzą w sekwencji, każdorazowe sprawdzenie spójności udostępnianego bloku może pozwolić węzłowi wycofać się z komunikacji, jeśli zostanie stwierdzone naruszenie spójności danych lokalnych. Ponieważ przebieg tej operacji nie jest uzależniony od przyjętego rozwiązania architektonicznego, dalej nie będzie analizowany.

Zakładając, że aplikacja użytkowa ma swobodny dostęp do bloków w łańcuchu, sprawdzenie poprawności jednego bloku wymaga sprawdzenia spójności całego łańcucha. Operacja ta ma złożoność O(N), gdzie N jest liczbą bloków w łańcuch. Można zatem przyjąć, że czas jej realizacji jest wprost proporcjonalny do ilości elementów. W konsekwencji możliwość wykonania takiej operacji całkowicie zależy od wymagań stawianych przez aplikację użytkową. Dodatkowo w przypadku współużywania API węzła procedura sprawdzania musiałaby otrzymać wzajemnie wyłączny dostęp do danych, aby zachować zasady sekcji krytycznej. Z tego powodu API oprogramowania węzła nie oferują takiej funkcjonalności. Jeśli jednak aplikacja użytkowa mająca swobodny dostęp do danych potrzebuje uzyskać gwarancję spójności może tą operację wykonać na własne potrzeby pobierając kolejne bloki począwszy od pierwszego i wybierając z łańcucha te, które są jej potrzebne.

Biorąc powyższe pod uwagę można stwierdzić, że w architekturze BlockGraph jest możliwe sprawdzanie spójności łańcucha przy wypakowywaniu grafów z bloków i zapisywaniu ich do repozytorium TripleStore. Nie jest natomiast możliwe sprawdzenie spójności przez aplikację korzystającą z klasy TripleStore bez dodatkowego dostępu do klasy Blockchain API w tej architekturze (patrz rozdział 3.2.2). Aby sprawdzenie spójności było możliwe musi dodatkowo być zaimplementowana funkcja porównywania grafów odczytanych z TripleStore z zawartości odpowiedniego bloku.

W architekturze GraphChain wszystkie dane zabezpieczające spójność łańcucha znajdują się w klasie TripleStore, a zatem każda aplikacja korzystająca z tych danych w uzasadnionych przypadkach może dokonać kontroli ich spójności przy odczycie poprzez sprawdzenie całego łańcucha grafów. W takim przypadku można zastosować następujący scenariusz:

* wykonanie dowolnej kwerendy lub ciągu kwerend
* sprawdzenie spójności repozytorium.

Jeśli po wykonaniu pewnej sekwencji kwerend repozytorium jest nadal spójne, to można przyjąć, że wyniki kwerendy są poprawne na poziomie gwarantowanym przez technologię Blockchain.

## Wydajność

Proponuje się, aby wydajność proponowanych rozwiązań mierzyć wydajnością kwerend SPARQL oraz wydajnością obliczeń kryptograficznej funkcji skrótu dla wybranych struktur danych. Ponieważ wszystkie proponowane rozwiązania eksponują dane za pośrednictwem typowego rozwiązania grafowej bazy danych, można stwierdzić, że czas ten nie zależy od przyjętej architektury z wyjątkiem BlockGraph i wykorzystaniu scenariusza Proxy (rozdział 3.2.2). W ostatnim przypadku świeża kopia wszystkich danych jest ładowana do bazy grafowej przed każdą kwerendą. Ponieważ złożoność tej operacji jest O(N) (gdzie N jest liczbą grafów) jej czas realizacji jest proporcjonalny do ilości danych. Ta własność sprawia, że rozwiązanie to nie jest praktyczne i dlatego nie jest rekomendowane do dalszych analiz związanych z wyborem rozwiązania wykorzystanego do realizacji dalszych prac.

## Elastyczność

Proponuje się dokonanie oceny proponowanych rozwiązań w aspekcie:

* ograniczeń wprowadzanych na protokół współdziałania węzłów tworzących sieć
* składni i semantyki zawartości bloków, a w tym wirtualnych bloków RDF
* różnorodność funkcji oferowanych aplikacji użytkowej.

W wyniku analiz przedstawionych rozwiązań architektonicznych można stwierdzić, że żadne z opisanych rozwiązań nie ogranicza swobody wyboru protokołu komunikacyjnego i mechanizmów uzyskiwania konsensusu.

W przypadku rozwiązania BlockGraph dane są dostępne również bezpośrednio z bloków tworzących łańcuch Blockchain. Dodatkowo autorska składnia i semantyka bloków czyni rozwiązanie otwartym z punktu widzenia semantyki i składni przechowywanych w blokach danych. W GraphChain dane są dostępne w formacie RDF. Rozwiązanie BlockGraph oferuje dodatkowo możliwość bezpośredniego dostępu do różnorodnych danych zapisanych w łańcuchu bloków. Z kolei, korzystając z uniwersalności reprezentacji RDF, możliwe jest zapisanie dowolnych danych w składowej „object” trójek za pomocą właściwej serializacji. Możliwy też jest mapowanie dowolnej struktury danych na graf jakim jest RDF.

# Podsumowanie

Biorąc pod uwagę analizę porównawczą proponowanych rozwiązań dokonaną w kontekście: bezpieczeństwa, wydajności i elastyczności przedstawioną w poprzednich rozwiązanych, do dalszych prac rekomendowana jest architektura **GraphChain**.

1. Craig Sayers, Alan H. Karp, „Computing the digest of an RDF graph”, HP Laboratories Palo Alto, 2004 [↑](#footnote-ref-2)
2. dla GB.cache i GC przyjęto, że porównanie grafów odbywa się metodą porównani policzonego skrótu [↑](#footnote-ref-3)
3. dla GB.proxy kontrola spójności na poziomie TS jest niepotrzebna [↑](#footnote-ref-4)
4. GC wstawiamy podwójną liczbę grafów [↑](#footnote-ref-5)