Politechnika Warszawska

WYDZIAŁ ELEKTRYCZNY

Instytut Elektrotechniki Teoretycznej
i Systemów Informacyjno-Pomiarowych
Zakład Elektrotechniki Teoretycznej
i Informatyki Stosowanej

Praca dyplomowa magisterska

na kierunku Informatyka Stosowana
w specjalności Inżynieria oprogramowania

Zastosowanie technologii Blockchain do weryfikacji obliczeń rozproszonych

Rafał Hofman
nr albumu 301133

promotor
prof. ndw. dr hab. inż. Michał Śmiałek

WARSZAWA 2020

Zastosowanie technologii Blockchain do weryfikacji obliczeń rozproszonych

Streszczenie

Celem poniższej pracy jest przedstawienie koncepcji użycia technologii Blockchain w systemie weryfikacji obliczeń rozproszonych, czyli takich, w których moc obliczeniowa maszyny może być udostępniana i przetwarzana przez różnych użytkowników. Praca zawiera wstęp teoretyczny nt. Blockchain, tłumaczący jego funkcje i główne cechy. Opisano zasady działania platformy Ethereum, którą wykorzystano do implementacji projektu. W rozdziale trzecim została omówiona technologia BalticLSC jako system obliczeń rozproszonych. Zawiera on opis architektury, diagramów oprogramowania oraz wymagań stawianych przed BalticLSC. W rozdziale czwartym omówiono projekt będący częścią praktyczną pracy magisterskiej i jego implementację. Zawarto w nim opis jednego ze sposobów weryfikacji obliczeń rozproszonych w BalticLSC na przykładzie wykorzystania modułów Blockchain. Rozdział piąty przedstawia wizję wykorzystania wyżej wymienionych modułów zaimplementowanych w projekcie w systemie BalticLSC celem lepszego zrozumienia jego składowych. Opisuje on proces wykonania obliczeń z wykorzystaniem projektu zaimplementowanego w części praktycznej pracy magisterskiej. Na koniec ujęto wnioski podsumowujące celowość pracy, lekcje wyciągnięte z implementacji projektu oraz dalsze perspektywy wykorzystania Blockchain w weryfikacji obliczeń rozproszonych.

Słowa kluczowe: Blockchain, BalticLSC, systemy rozproszone, obliczenia rozproszone, werfykacja obliczeń

Usage of Blockchain technology for verification of distributed computing

Abstract

The thesis aims to show the concept of the Blockchain usage for verification of distributed computations data. That is such data that can be shared and processed by different users. The thesis consists of a theoretical introduction to the Blockchain, describing its function and main characteristics. Ethereum technology is described, being chosen as a Blockchain example for the implementation of the project. Chapter three describes the BalticLSC distributed computations data system that includes its software diagrams, architecture, and system requirements. The project implemented as a practical part of the thesis is described in chapter 4. It consists of the proposal of verification of BalticLSC distributed computations data using the Blockchain technology. Chapter five is describing proof of concept of using the implemented solution together with the BalticLSC distributed computations data. The last chapter sums up achievements from the thesis, lessons learned, and future perspectives of the Blockchain usage for verification of distributed computations data.

Keywords: Blockchain, BalticLSC, distributed systems, distributed computations, computations verification

WARSZAWA, 22 czerwca 2020

POLITECHNIKA WARSZAWSKA
WYDZIAŁ ELEKTRYCZNY

OŚWIADCZENIE

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa magisterska pt. Zastosowanie technologii Blockchain do weryfikacji obliczeń rozproszonych:

- została napisana przeze mnie samodzielnie,
- nie narusza niczyich praw autorskich,
- nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam, że przedłożona do obrony praca dyplomowa nie była wcześniej podstawą postępowania związanego z uzyskaniem dyplomu lub tytułu zawodowego w uczelni wyższej. Jestem świadom, że praca zawiera również rezultaty stanowiące własności intelektualne Politechniki Warszawskiej, które nie mogą być udostępniane innym osobom i instytucjom bez zgody Władz Wydziału Elektrycznego.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Rafał Hofman.....

Spis treści

Rozdział 1

Wstęp

1.0.1 Motywacja i cel pracy

Obecnie wiele systemów opartych jest na użyciu technologii Blockchain. Sieci te zdobyły popularność m.in. ze względu na bezpieczeństwo oraz stabilność zawartych w nich danych, co odgrywa szczególną rolę w systemach transakcyjnych. Celem poniższej pracy jest przedstawienie koncepcji użycia technologii Blockchain w systemie weryfikacji obliczeń rozproszonych, czyli takich, w których moc obliczeniowa maszyny może być udostępniana i przetwarzana przez różnych użytkowników. Do takich systemów należy BalticLSC, na podstawie którego stworzono projekt będący częścią praktyczną niniejszej pracy. Celem projektu jest próba implementacji walidacji obliczeń rozproszonych za pomocą technologii Blochchain. Pozwoli to uzyskać odpowiedź na pytanie, czy technologia Blockchain jest odpowiednia do tego typu zastosowań.

1.0.2 Treść pracy

Użytkowników systemu BalticLSC możemy podzielić na tych, którzy wykonują obliczenia na cudzych maszynach oraz na takich, którzy oferują własną maszynę do przeprowadzania obliczeń rozproszonych. Technologia Blockchain, której istotę objaśniono w rozdziale drugim, mogłaby zostać sprawdzona i wykorzystana do zweryfikowania tożsamości podmiotu, który wykonał obliczenia w systemie BalticLSC.

Rozdział drugi zawiera wstęp teoretyczny nt. Blockchain, tłumacząc jego funkcje i główne cechy. W kolejnym podrozdziale opisano zasady działania platformy Ethereum, którą wybrano do implementacji projektu. Przy wyborze technologi kierowano się jej wysoką popularnością.

W dalszej części rozdziału omówiono ważne w automatyzacji procesów

Smart Contracts, które należą do jednych z narzędzi Blockchain. Opisano możliwości zastosowania ich w platformie Ethereum. Następnie przybliżono sposoby tworzenia nowych bloków w Ethereum oraz włączania ich do Blockchain. Zostały również wyjaśnione pojęcia tj. *Blockchain consensus*, *Proof of Work*, *Proof of Stake* oraz *Proof of Authority* ważne ze względu na to, że definiują one tworzenie nowych bloków danych. Technologia Blockchain najczęściej oparta jest o rejestr publiczny. W części praktycznej pracy wykorzystano jednak prywatny łańcuch. Z tego powodu w dalszej części rozdziału podjęto istotny temat prywatności w sieci Ethereum oraz pojęcie prywatnych łańcuchów. W dalszej kolejności opisano Golem jako wybraną do projektu jedną z implementacji obliczeń rozproszonych wykorzystującą technologię Etherum.

W rozdziale trzecim została omówiona technologia BalticLSC jako system obliczeń rozproszonych. Przedstawiono tutaj dokument *enviorement vision* projektu BalticLSC, który opisuje wymagania odnośnie systemu. Następnie umieszczono zwięzły opis architektury platformy BalticLSC. Ze względu na rozbudowane rozmiary diagramów użytych w technologii BalticLSC oraz brak możliwości ich czytelnego umieszczenia w pracy, w ostatnim podrozdziale opisano tylko najważniejsze pojęcia w kontekście tej pracy, prezentując fragmenty diagramów.

W rozdziale czwartym omówiono projekt będący częścią praktyczną pracy magisterskiej i jego implementację. Zawarto w nim opis jednego ze sposobów weryfikacji obliczeń rozproszonych w BalticLSC na przykładzie wykorzystania modułów Blockchain. W tym celu wykorzystano fragmenty diagramów BalticLSC z usytuowanymi komponentami Blockchain. W dalszej części opisano technologie użyte do implementacji projektu. Następnie umieszczono opis stworzonych modułów *Blockchain*, *Blockchain service*, *Blockchain signer* oraz *Blockchain viewer*, które zostały zaimplementowane w ramach pracy magisterskiej. Do opisów modułów projektu celem przekazania jego istoty w sposób klarowny dołączono diagramy oraz fragmenty ich klas.

Rozdział piąty przedstawia wizję wykorzystania wyżej wymienionych modułów zaimplementowanych w projekcie w systemie BalticLSC celem lepszego zrozumienia jego składowych. Opisuje on proces wykonania obliczeń z wykorzystaniem projektu zaimplementowanego w części praktycznej pracy magisterskiej. Proces ten rozpoczyna się wykonaniem obliczeń, a następnie trwa do momentu zweryfikowania ich przez system z wykorzystaniem technologii Blockchain.

Na koniec ujęto wnioski podsumowujące celowość pracy, lekcje wyciągnięte z implementacji projektu oraz dalsze perspektywy wykorzystania Blockchain w weryfikacji obliczeń rozproszonych.

Warto wspomnieć, że z uwagi na małą liczbę polskich publikacji nt. technologii Blockchain, w wielu przypadkach nie odnaleziono tłumaczenia anglo-

językcznej terminologii na język polski. Z tego powodu zdecydowano o pozostawieniu tych pojęć w języku angielskim.

Dodatek 1 opisuje komendy jakie należy wykonać, aby uruchomić obrazy z modułami *Blockchain*, *Blockchain service*, *Blockchain signer* oraz *Blockchain viewer* rozwijanymi w ramach pracy magisterskiej. Dodatek 2 opisuje REST API wykorzystywane przez moduły *Blockchain*, *Blockchain service*, *Blockchain signer* oraz *Blockchain viewer* wykonane w ramach pracy magisterskiej.

Podsumowując, praca składa się ze:

1. wstępu teoretycznego nt. systemu Blockchain, tłumacząc jego funkcje i główne cechy, jako przykład opisując technologię Ethereum
2. wstępu teoretycznego nt. systemu obliczeń rozproszonych BalticLSC, jego wizji oraz diagramów i pojęć istotnych w kontekście pracy dyplomowej
3. informacji o zaimplementowanym projekcie - w tym o dopasowaniu obecnych komponentów BalticLSC do użycia Blockchain oraz opisie poszczególnych serwisów projektu - *Blockchain service*, *Blockchain signer* oraz *Blockchain viewer*
4. studium przypadku wykorzystującego zaimplementowany projekt do weryfikacji obliczeń rozproszonych BalticLSC z użyciem Blockchain
5. wniosków zawierających wady i zalety powyższego rozwiązania, możliwości wykorzystania projektu oraz korzyści edukacyjnych dla autora tej pracy
6. dokumentacji obrazów zaimplementowanych w ramach projektu oraz REST API poszczególnych serwisów

Rozdział 2

Technologia Blockchain

2.1 Wstęp do technologii Blockchain

Żeby zrozumieć czym jest technologia Blockchain, należy rozbić tę frazę na dwie części:

- *Block* z języka angielskiego blok - czyli w tym przypadku zestaw danych
- *Chain* z języka angielskiego łańcuch - w tym przypadku łańcuch bloków

Blockchain jest zatem blokiem danych połączonym łańcuchem. Z danych zawartych w bloku wyliczana jest funkcja skrótu. Jest ona przechowywana w nagłówku bloku. Następny blok w łańcuchu ma odnośnik do nagłówka poprzedniego bloku.

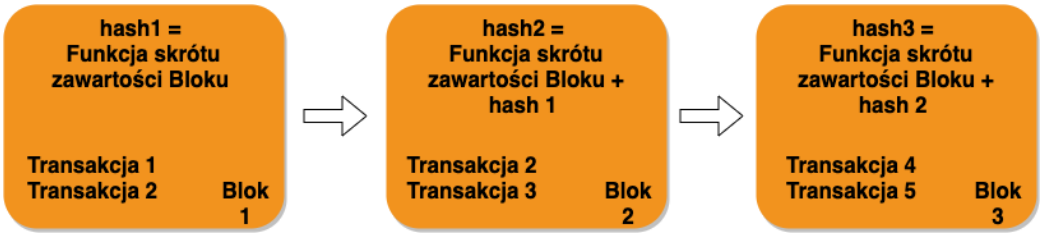
Zasady działania Blockchain przedstawiono na rysunku 2.1.

Każda zmiana danych w Blockchain spowodowałaby zmianę nagłówka bloku, z którym połączone są kolejne bloki. W efekcie zachodzi konieczność przeliczenia nagłówków każdego z następnych bloków w sieci. Tymczasem po zapełnieniu powstałych bloków sieć aktywnie dodaje do łańcucha kolejne na podstawie najdłuższego łańcucha bloków. Taki mechanizm zapewnia bezpieczeństwo danych w sieci Blockchain.

Istnieją także interaktywne aplikacje prezentujące działanie sieci Blockchain [4].

Blockchain jako technologia charakteryzuje się [2] [5]:

- połączeniem *peer to peer* - (P2P)- maszyny znajdujące się w sieci o tej architekturze połączone są ze sobą w taki sposób, że nie ma centralnego serwera, którego wyłączenie spowodowałoby wyłączenie całej sieci.
- zdecentralizowaną i rozproszoną bazą danych [5] - wszystkie maszyny podłączone do sieci mogą mieć różną lokalizację i przechowują dane.



Rysunek 2.1: Jak działa Blockchain

Każda zmiana danych dokonana w jednej maszynie pociągnie za sobą również zmianę danych w innych maszynach tego systemu. Wypadnięcie z takiej sieci jednego komputera nie spowoduje więc utraty danych.

- użyciem algorytmów kryptograficznych - funkcja skrótu jak i kryptografia asymetryczna jest powszechnie wykorzystywana w sieci.

2.2 Technologia Ethereum jako przykład Blockchain

2.2.1 Wstęp do technologi Ethereum

Blockchain jest technologią używaną głównie do przesyłania cyfrowych pieniędzy, które nazywamy kryptowalutami. Umożliwiają one dokonywanie płatności na całym świecie bez udziału banku i innych osób. Przykładem jest Ether - kryptowaluta obsługiwana przez Ethereum - jedną z wiodących platform walutowych wykorzystujących Blochchain.

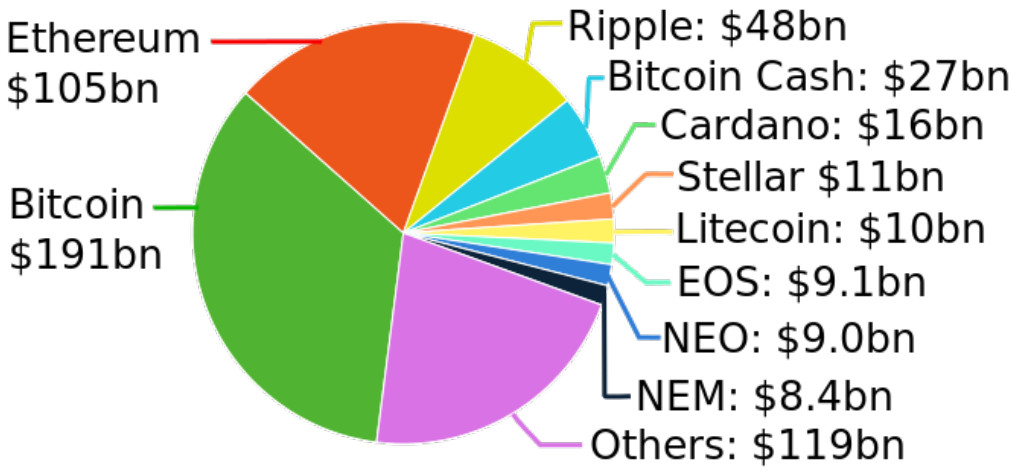
Popularność Ethereum ukazuje rysunek 2.2. Waluta Ethereum - Ether - plasuje się na drugim miejscu, zaraz po Bitcoinie.

Najpopularniejsze rozwiązania do tworzenia prywatnych sieci Blockchain to Ethereum, Hyperledger i Corda. Jak widać z analizy częstości wyszukiwania w Google na rysunku 2.3, hasło zawierające Ethereum było najbardziej popularnym z powyższych.

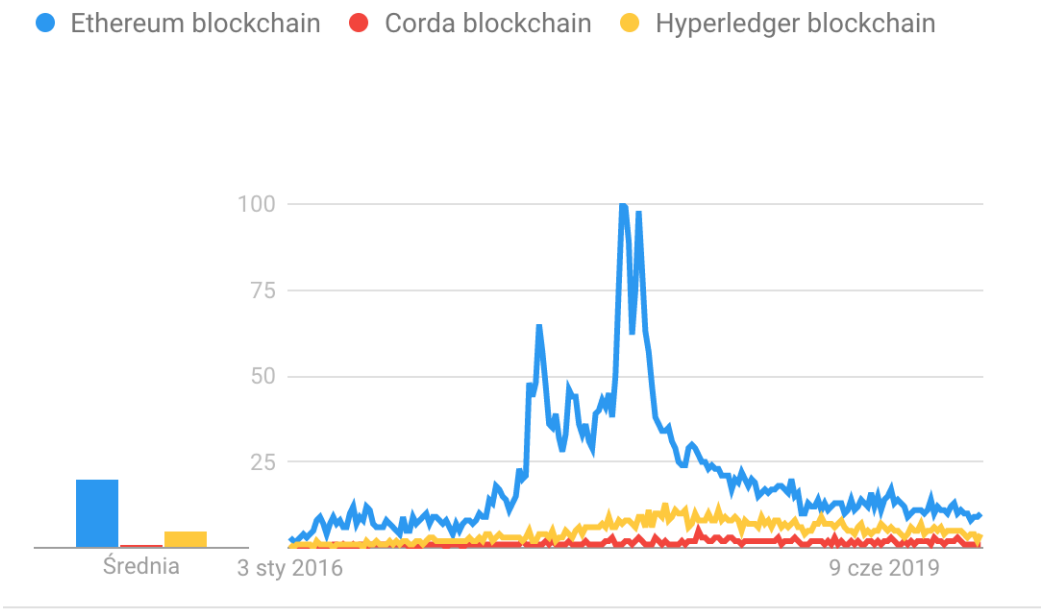
Poza obrotem kryptowalutami Ethereum pozwala na używanie własnej logiki biznesowej. Można ją osadzić na łańcuchu Blockchain za pomocą *Smart Contract*.

2.2.2 Smart Contracts

Smart Contract to inaczej kod, który zostaje zainicjalizowany na Blockchainie. Inicjalizacja odbywa się przez kompilację *Smart Contract* do *byte-*

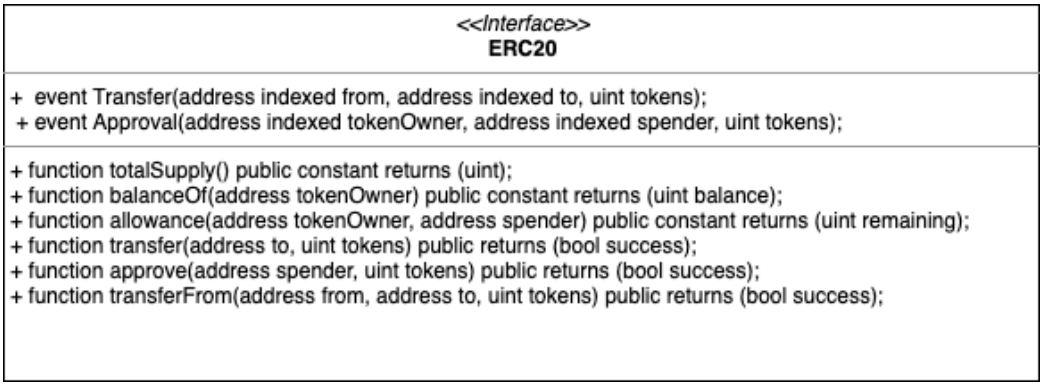


Rysunek 2.2: Popularność kryptowalut [8]



Rysunek 2.3: Google - popularność wyszukiwanych haseł [7]

code, a następnie umieszczenie kodu na łańcuchu. Kod reprezentuje pewną logikę biznesową, która trafi na łańcuch. W praktyce kod ten jest odpowiedni-



Rysunek 2.4: Diagram interfejsu ERC20 [11]

kiem pewnego rodzaju inteligentnej umowy narzucającej użytkownikom warunki, jakie muszą spełnić.

W projekcie będącym częścią praktyczną niniejszej pracy do napisania *Smart Contract* użyto Solidity [9]. Jest on popularnym językiem do tworzenia *Smart Contract* w Ethereum

Szeroko wykorzystywanym interfejsem *Smart Contract* w Ethereum jest ERC20. Służy on do stworzenia własnej kryptowaluty. Diagram interfejsu jest widoczny na rysunku 2.4.

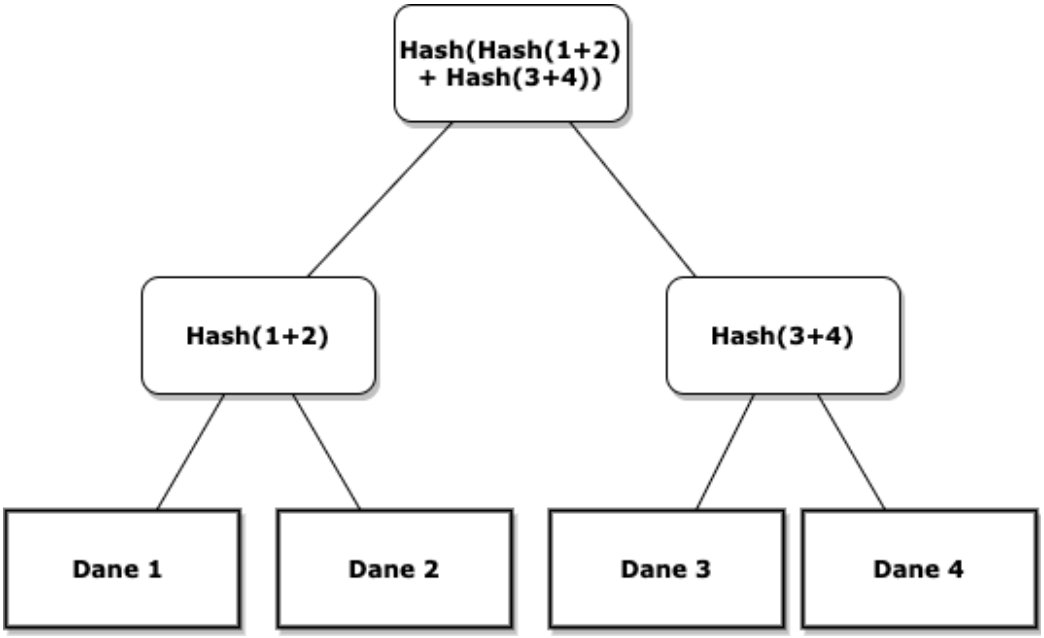
Smart Contract ERC20 przechowuje dane o właścicielach waluty i wartości ich kont. Waluta może być przelewana między użytkownikami za pomocą metody *transfer*. Przykładem implementacji ERC20 jest Dodge Coin. Jego *Smart Contract* oraz wszystkie transakcje z wykorzystaniem Dodge Coin są dostępne na przeglądarce transakcji Ethereum [10].

2.2.3 Zasada działania Ethereum

Podstawowa zasada działania Ethereum jest oparta o koncept *accounts*. *Accounts* to klucze: prywatny i publiczny. Klucz publiczny to taki, do którego każdy ma dostęp (odpowiednik numeru konta bankowego). Z klucza publicznego derywowany jest *adress*. Służy on do identyfikacji konta

Konto może być kontrolowane przez człowieka (adres osoby) lub *Smart Contract* (adres *Smart Contract*). Aby zmienić stan Blockchain, konto wysyła *transaction*. *Transaction* jest podpisana przez klucz prywatny danego konta (odpowiednik PINu/hasła). *Transaction* wysyłana w Ethereum jest wykonywana na maszynie wirtualnej znanej jako *Ethereum Virtual Machine*.

W momencie wysłania transakcji na łańcuch *Ethereum Virtual Machine* dokonuje obliczeń ilości wykonanych przez daną transakcję operacji oraz ilo-



Rysunek 2.5: *Merkle Patricia Tree* w Ethereum

ści zmian w stanie Blockchain. Opłata za transakcję jest sumowana w *gas*, który jest miarą mocy obliczeniowej. Ostatecznie przeliczany zostaje on na walutę *Ether*. Celem płatności *gas* jest ograniczenie ataku DDOS (Distributed Denial of Service) na sieć Blockchain.

Transaction po udanym wykonaniu zostaje włączona do bloku. Od tej chwili *transaction* stanowi cześć Blockchain. Dzięki temu, że *transaction* jest podpisana, można łatwo stwierdzić, z jakiego *adress* wysłano *transaction*.

Stan Blockchain w Ethereum reprezentowany jest za pomocą *Merkle Patricia Tree*. Jego przykład przedstawiono na rysunku 2.5. *Merkle Patricia Tree* to drzewo hash-rodzaj przechowywania i przetwarzania danych w pamięci maszyny, dzięki któremu możliwa jest bezpieczna weryfikacja danych przekazywanych między komputerami. Za pomocą *Merkle Patricia Tree* przechowuje się w Ethereum stan kont czy transakcji. Kolejne liście *Merkle Patricia Tree* stanowią wynik funkcji skrótu dwóch liści pod nim. Ostatecznie wynik funkcji zawartości całego drzewa jest przechowywany w liściu górnym. Wartość liścia górnego *Merkle Patricia Tree* wchodzi w zawartość nagłówka bloku. Dzięki temu zmiana w jakimkolwiek liściu w *Merkle Patricia Tree* spowoduje zmianę liścia górnego *Merkle Patricia Tree*, a tym samym nagłówka bloku.

2.2.4 Tworzenie nowych bloków

Jeśli przebieg *Transaction* jest udany, zostaje ona włączona do bloku. Najpierw należy jednak uzgodnić jaki jest *Blockchain consensus* dla sieci. *Blockchain consensus* warunkuje, kto i w jakim czasie może dodać nowy blok do łańcucha. Opisuje również sposób dodawania nowych bloków do sieci.

Popularne *Blockchain consensus* w Etereum:

Proof of Work - jest najbardziej znanym aktywnym na publicznej sieci Ethereum wśród wszystkich *Blockchain consensus*.

Tworzenie bloku w *Proof of Work* polega na znalezieniu pewnej liczby i dodaniu jej do nagłówka bloku. Liczba ta nazwana *nonce* ma być mniejsza lub równa:

$$n \leqslant \frac{2^{256}}{diff} \tag{2.1}$$

gdzie:

- n definiuje *nonce* - czyli liczbę szukaną
- diff definiuje *difficulty* - czyli trudność sieci, która jest dostosowywana.

W celu znalezienia tej liczby używany jest algorytm *Ethash* [6]. Ze względu na to, że znalezienie tej liczby wymaga wielokrotnego użycia algorytmu *Ethash*, *Proof of Work* wymaga użycia dużych mocy obliczeniowych uzyskiwanych np. przez wiele procesorów GPU. Stworzenie nowego bloku łączy się z pewną nagrodą pieniężną w Ether.

Proof of Stake - działa podobnie do loterii [12]. Użytkownik, który posiada więcej waluty Ether, ma większą szanse wygrać w loterii, w której nagrodą jest stworzenie bloku.

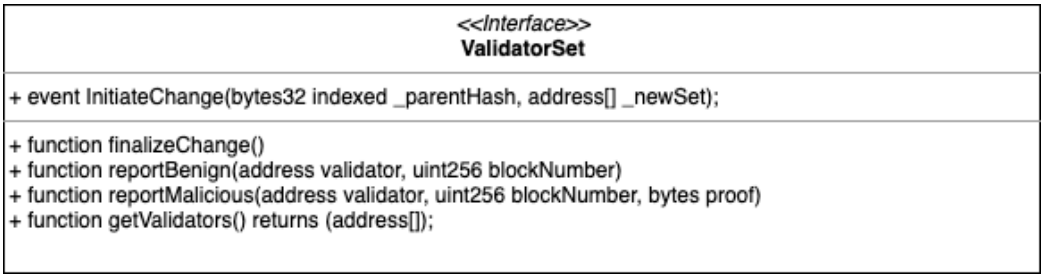
Podobnie jak w *Proof of Work*, stworzenie nowego bloku łączy się z pewną nagrodą pieniężną w Ether.

Obecnie główna sieć Ethereum jest w trakcie przekształcania na *Proof of Stake* w aktualizacji Casper [12].

Proof of Authority - tożsamość osoby/organizacji dodającej nowy blok jest gwarancją dla pozostałych uczestników w łańcuchu.

W *Proof of Authority* nie ma pieniężnej nagrody za dodanie bloku. Jako że w *Proof of Authority* dodający nowe bloki jest znany sieci, można skonfigurować sieć do swoich potrzeb. Pozwala to na skonfigurowanie sieci np. do szybkiego i darmowego dodawania nowych transakcji.

Implementacją *Proof of Authority* jest algorytm Aura [15].



Rysunek 2.6: Diagram interfejsu ValidatorSet

Algorytm Aura [15] działa w prosty sposób. Nadaje on czas systemowy, w której każdy *validator* może dodawać blok do sieci. *Validator* to inaczej osoba uprawniona przez sieć do dodawania nowych bloków.

Startując Blockchain, *validator* można wyspecyfikować poprzez:

- listę ich *adress*
- *Validator Contract*

Validator Contract pozwala na dodawanie czy usuwanie *validator* w czasie działania sieci. Aby zastosować *Validator Contract*, należy stworzyć *Smart Contract*, implementując odpowiedni interfejs [16]. Jego diagram jest widoczny na rysunku 2.6.

2.2.5 Prywatność w sieci Ethereum

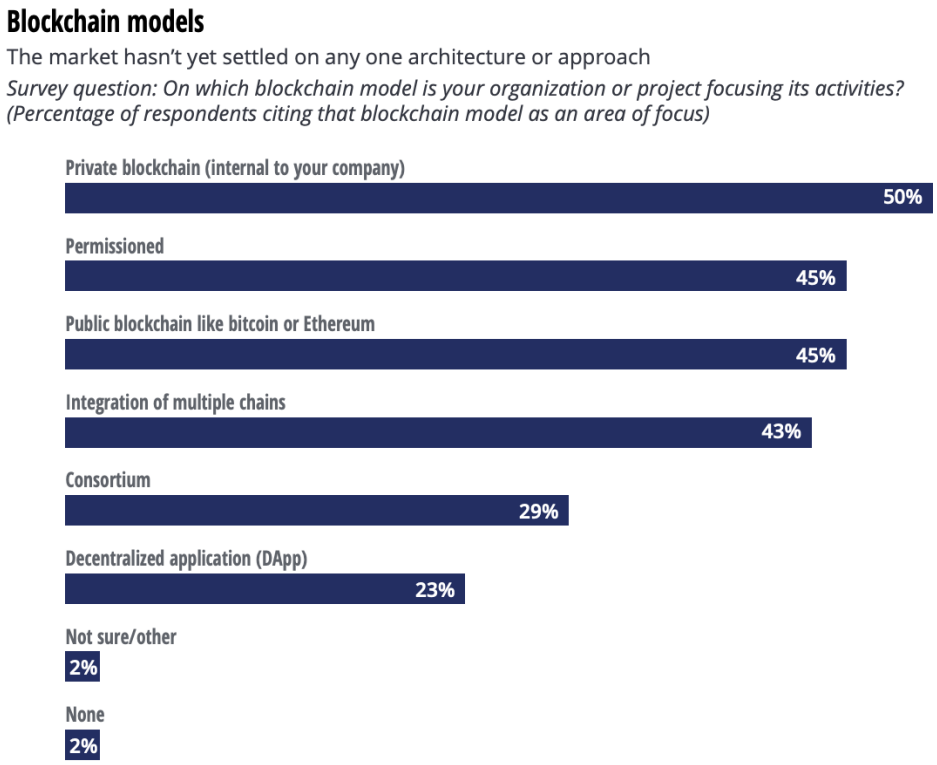
Dane dostępne na głównym łańcuchu Ethereum są publiczne. Oznacza to, że dowolna osoba może odczytać zarówno stan Blockchain jak i dane, które na niego się składają. Nawet jeżeli *Smart Contract* jest obecny w łańcuchu w formie *bytecode*, dostępne są narzędzia do inżynierii wstecznej celem poznania jego interfejsu [13].

Obecnie powstają opracowania prywatności danych wykorzystujące główną sieć Ethereum [14]. Są to jednak rozwiązania eksperymentalne.

Warunkiem zapewnienia prywatności i ochrony danych jest stworzenie prywatnej sieci Blockchain.

Przedstawiony na 2.7 fragment raportu Deloitte 2019 ukazuje popularność sieci prywatnych. Świadczy to o dużej ilości rozwiązań opartych o prywatne łańcuchy Blockchain.

Ze względu na koszt transakcji na łańcuchu publicznym oraz na wrażliwość danych obliczeniowych (chęć ograniczenia publicznego dostępu do danych) zdecydowano użyć prywatnej sieci Blockchain. W projekcie magister-



Rysunek 2.7: Fragment raportu Deloitte [3]

skim użyty jest zatem *Blockchain consensus Proof of Authority*, którego implementacją jest algorytm Aura [15], co gwarantuje spełnienie powyższych warunków.

2.2.6 Użycie Ethereum do obliczeń rozproszonych

Jednym z wielu zastosowań biznesowych Platformy Ethereum jest wykorzystanie jej do obliczeń rozproszonych.

Przykładem implementacji dotyczącej obliczeń rozproszonych z wykorzystaniem Ethereum jest Golem [17]. Przy wyborze projektu uwzględniono jego polskie pochodzenie oraz współistniejącą współpracę z konsorcjum BalticLSC.

Golem wykorzystuje platformę Ethereum do obracania wirtualną walutą służącą do zapłaty za obliczenia rozproszone. Dzieli on podmioty na dwa rodzaje:

- *Provider* - udostępnia swoją maszynę do obliczeń rozproszonych. Otrzy-

muje on za to odpowiednią zapłatę poprzez walutę internetową Golem.

- *Requestor* - wykorzystuje dostępne maszyny do obliczeń rozproszonych. Płaci on za wykorzystane zasoby.

W tym przypadku waluta internetowa (kryptowaluta), za pomocą której wykonywane są obliczenia rozproszone to GNT. Jest to waluta zastosowana w Ethereum oparta o interfejs ERC20, o którym była mowa w poprzednich podrozdziałach.

Rozdział 3

BalticLSC - system obliczeń rozproszonych

3.1 Wstęp do projektu BalticLSC

BalticLSC (*Baltic Large Scale Computing*) jest projektem obliczeń rozproszonych współprowadzonym przez kilku partnerów Europejskich. Projekt ten jest współfinansowany przez Interreg należący do Unii Europejskiej [30].

W sieci BalticLSC można kupować oraz oferować własną moc obliczeniową. Celem BalticLSC jest udostępnienie mocy obliczeniowej średnim i małym przedsiębiorstwom [30].

Innowacyjne cele biznesowe, takie jak projektowanie statków, wymagają dostępu do dużych mocy obliczeniowych. Są one obecnie zarezerwowane dla większych firm, które ze względu na fundusze mogą sobie na to pozwolić. BalticLSC łączy grupy przedsiębiorców, którzy mogą się dzielić mocą obliczeniową za opłatą.

3.2 Wymagania dla systemu BalticLSC

Wizja środowiska projektu jest zawarta na stronie BalticLSC [32]. Stanowi ona wymagania projektu stawiane przez użytkowników końcowych. W roli interesariuszy projektu dokument stawia m.in. instytucje publiczne, organizacje, stowarzyszenia badawcze oraz małe i średnie przedsiębiorstwa. W dokumencie przedstawione są głównie problemy interesariuszy takie jak np. brak sprzętu do obliczeń rozproszonych lub brak odpowiednich danych. W następnej kolejności dokument opisuje cechy produktu, w tym jego cechy funkcjonalne np. zarządzanie obliczeniami rozproszonymi lub rozwój aplikacji rozproszonych. Wizja środowiska podaje także niefunkcjonalne cechy

produktu takie jak integracja z zewnętrznymi środowiskami czy odległość fizyczna od danych obliczeniowych. Pełna lista powyższych wymagań znajduje się w dokumencie. Przedstawiono w nim również studium przypadku - zaproponowano i opisano tutaj sytuacje, w których system BalticLSC mógłby zostać wykorzystany do obliczeń danych rozproszonych. Propozycje składają się m.in. z frameworka, który ma zastosowanie w przetwarzaniu danych wideo lub modelowaniu molekularnym.

3.3 Architektura BalticLSC

Wizja architekturalna platformy jest zawarta na stronie BalticLSC [33]. Komponenty BalticLSC są skonteneryzowane w Docker [25]. Moduły w Docker pozostają niezależne od środowiska, w którym są użytkowane. Dodatkowo, można je poddawać automatyzacji czy skalowaniu, do czego jest wykorzystywany Kubernetes [34]. W celu zarządzania Kubernetesem stosuje się system Rancher [35]. W BalticLSC służy on także do monitorowania i zarządzania zasobami maszyny.

Użytkownicy systemu BalticLSC są izolowani, a zasoby przez nich wykorzystane zostają limitowane. Odpowiednio wyizolowani użytkownicy tworzą swoje projekty uruchamiane przez Ranchera. W Rancherze istnieje możliwość wyizolowania przestrzeni nazw. Nie ma natomiast opcji przydzielania jej do projektu stworzonego przez użytkownika. W tym przypadku używa się *Platform Manager*, który jest proxy do Ranchera [33]. Ma on za zadanie przechwytywać żądania sieciowe do Ranchera oraz łączyć przestrzenie nazw z użytkownikiem.

Komponenty BalticLSC kontaktują się przez REST API. Sieć BalticLSC jest odpowiednio izolowana od innych aplikacji działających na maszynie. Kubernetes może tego dokonać poprzez tzw. *network polices* [33]. Całość architektury platformy BalticLSC przedstawiono na rysunku 3.1.

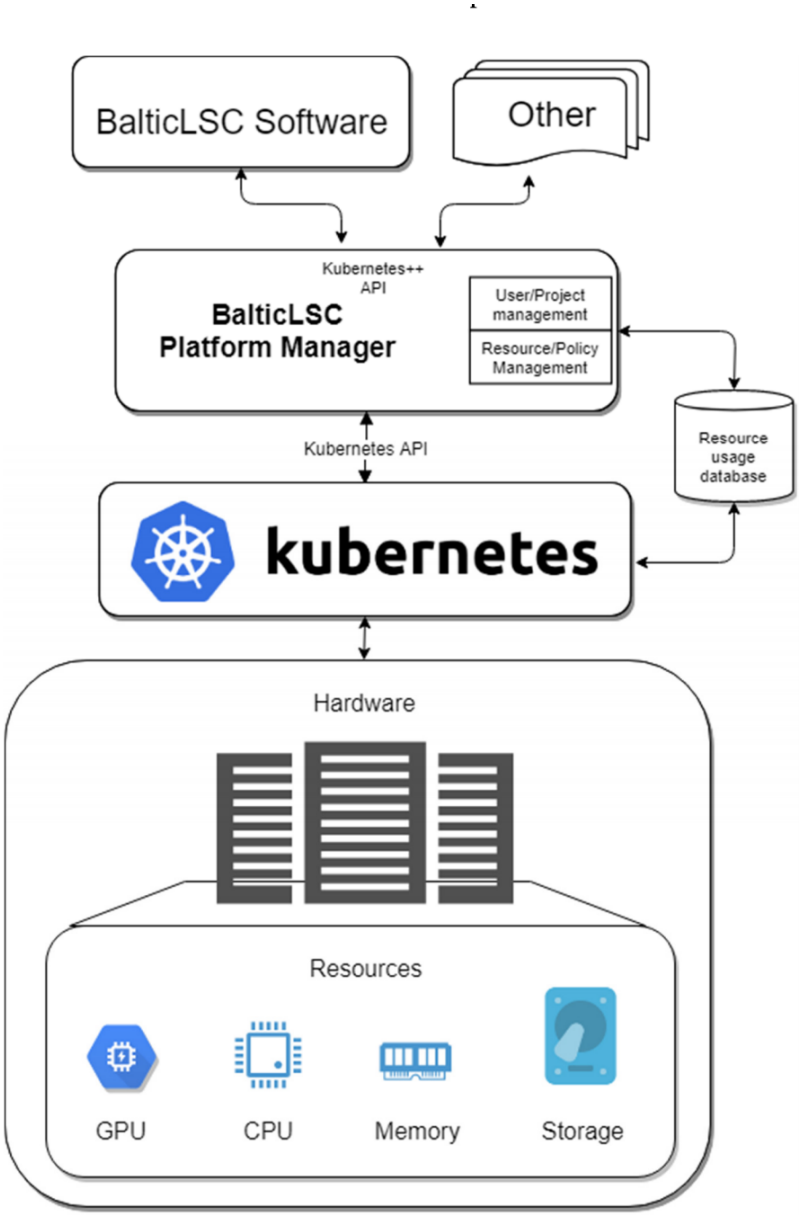
Wizja architektury oprogramowania projektu jest zawarta na stronie BalticLSC [36].

BalticLSC może mieć jeden lub więcej *Master node*. *Master node* jest częścią BalticLSC, która zawiera dane o użytkownikach, zasobach czy aplikacjach rozproszonych BalticLSC. *Master node* wykorzystuje *BalticLSC Registry* do przechowywania danych.

Resource clusters to inaczej klastry Kubernetes z zainstalowanym oprogramowaniem do przeprowadzenia obliczeń rozproszonych BalticLSC. *Resource clusters* są zarejestrowane w *Master node*.

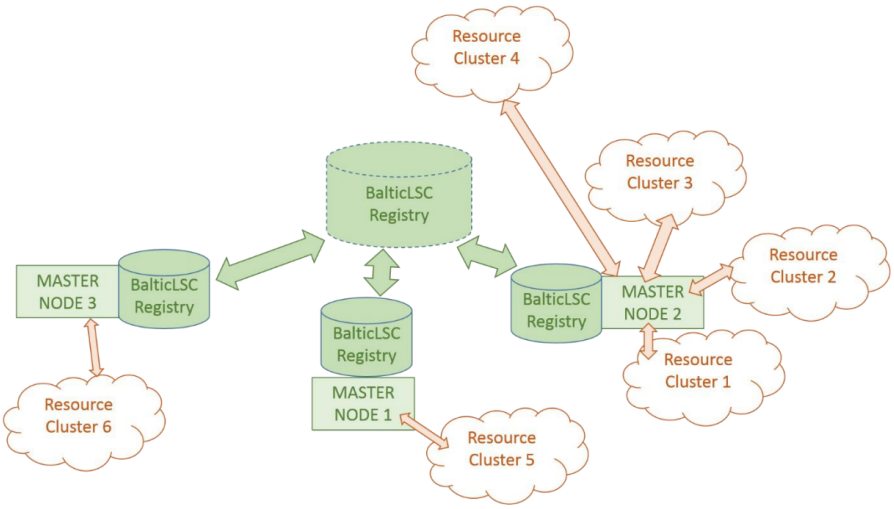
Architekturę oprogramowania widać na rysunku 3.2.

Master node składa się z:



Rysunek 3.1: Architektura w systemie BalticLSC [31]

- *Registry* - służącego jako miejsce do przechowywania danych
- *Admin tool* - służącego do zarządzania zasobami BalticLSC
- *App store* - służącego do tworzenia aplikacji BalticLSC
- *Computation tool* - służącego do zarządzania *Resource clusters*



Rysunek 3.2: Architektura oprogramowania w systemie BalticLSC [36]

BalticLSC ma architekturę mikroserwisową. Pozwala to na sprawniejsze zarządzanie zespołem developerskim oraz skalowalność serwisów. Architektura mikroserwisowa pozwala na użycie dowolnej technologii, ponieważ kod jest skonteneryzowany, a serwisy zorkiestrowane.

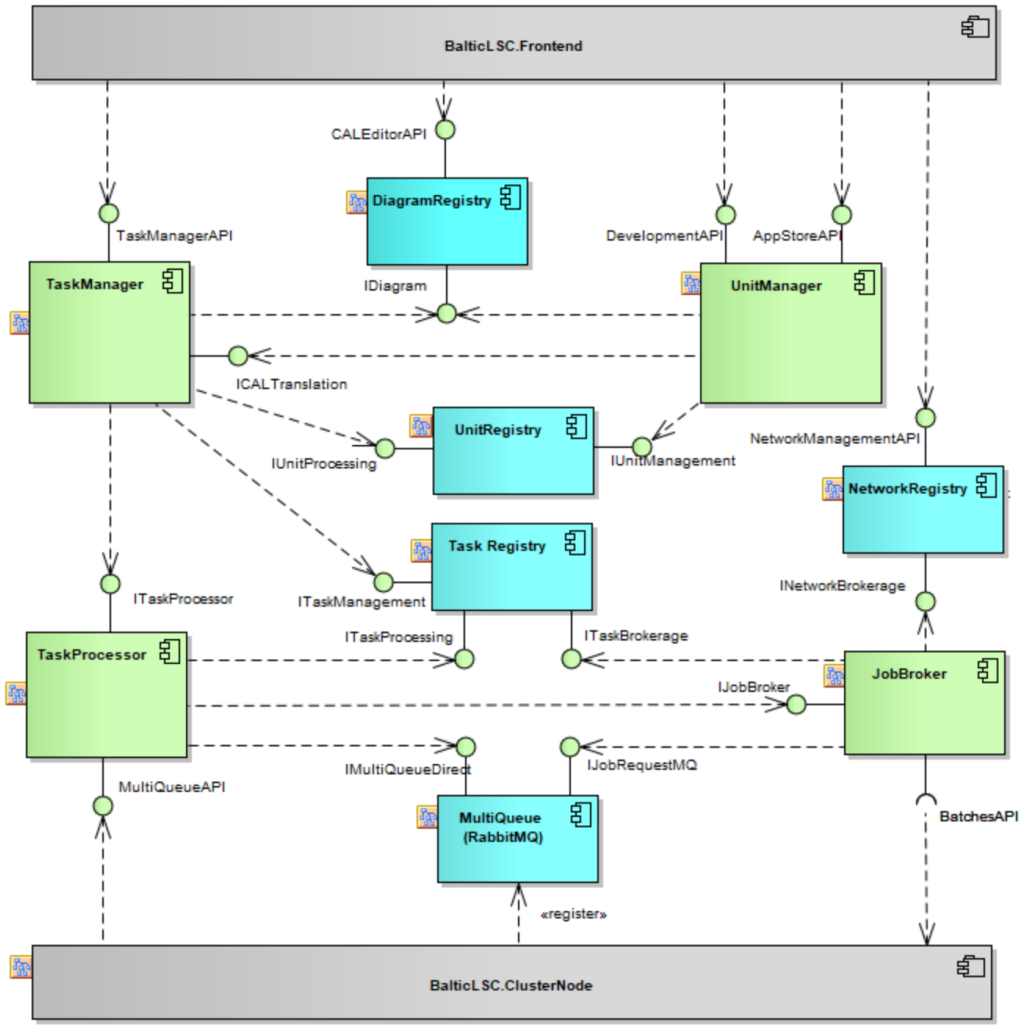
3.3.1 Diagramy oprogramowania BalticLSC

Pojęcia dotyczące BalticLSC istotne w kontekście pracy magisterskiej można wytłumaczyć wykorzystując wizję rozwoju oprogramowania [36] oraz diagramy [37].

BalticLSC składa się na część *Frontend*, *Master Node Backend* oraz *Cluster Node*.

Część *Frontend* jest odpowiedzialna za interakcję z użytkownikiem.

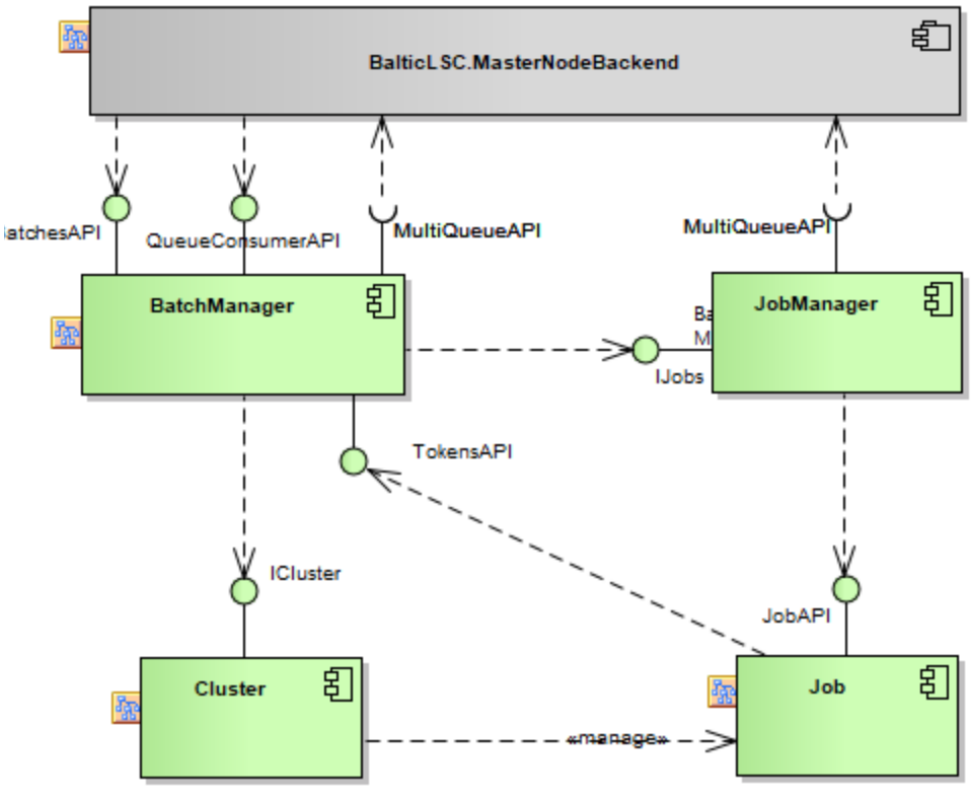
Rysunek 3.3 przedstawia diagram komponentów *Master Node Backend*. Jego główną funkcją jest zarządzanie obliczeniami BalticLSC. Składa się on z rejestrów zawierających informacje o poszczególnych zadaniach obliczeniowych. Należą do nich *Diagram Registry*, *Unit Registry*, *Task Registry* i *Network Registry*. *Master Node Backend* zleca także zadania obliczeniowe dla poszczególnych *Cluster Node* (czyli maszyn obliczeniowych uczestniczących w systemie). Odbywa się to poprzez kolejkę *MultiQueue (RabitMq)*, a podmiotem zlecającym jest *Job Broker*. Komponenty takie jak *Unit Manager* czy *Task Manager* odpowiadają za zarządzanie maszynami obliczeniowymi i zadaniami.



Rysunek 3.3: Diagram komponentów *Master Node Backend* [37]

Rysunek 3.4 przedstawia diagram komponentów *Cluster Node*. *Cluster Node* to inaczej maszyna uczestnicząca w systemie BalticLSC, która wykonuje obliczenia. Składa się ona z oprogramowania BalticLSC obecnego na danym *Cluster Node*, które przekazuje i zleca konkretne zadania obliczeniowe za pomocą wiadomości. *Cluster Node* zawiera także *Computation Cluster* czyli klaster obliczeniowy wykonujący obliczenia. Zadanie obliczeniowe są zlecane klastrowi przez *Batch Manager* i *Job Manager* - komponenty oprogramowania BalticLSC obecnego na *Cluster Node*.

Rysunek 3.5 przedstawia słownik *Batch Manager*. *Batch Manager* jest

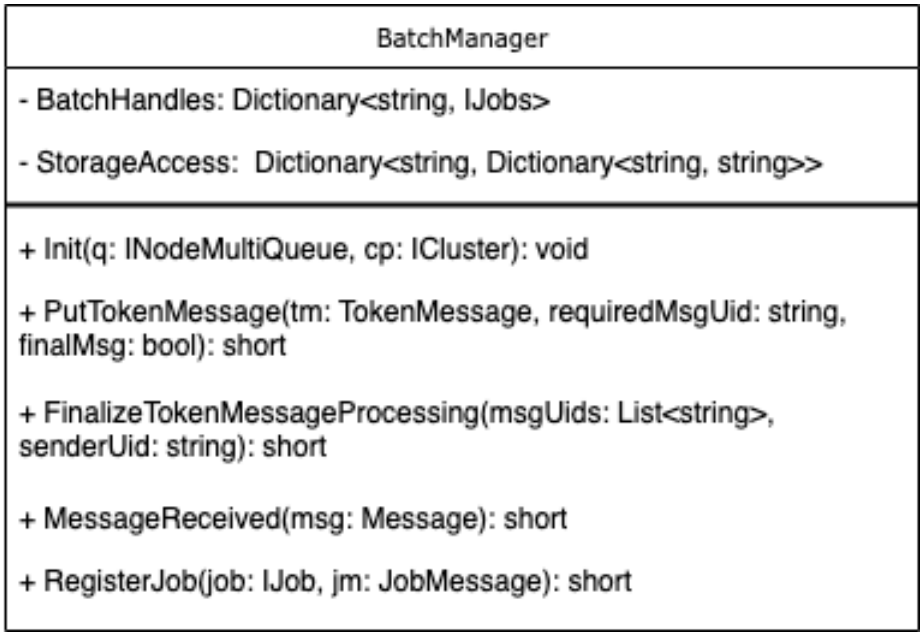


Rysunek 3.4: Diagram komponentów *Cluster Node* [37]

częścią oprogramowania BalticLSC zainstalowaną na *Computational Resource*. Zleca on zadania obliczeniowe klastrowi *Computation Cluster* znajdującemu się na tej maszynie.

Rysunek 3.6 przedstawia słownik *Computation Resource*. Dzięki niemu można się dowiedzieć, w jaki sposób w BalticLSC zamodelowana jest maszyna wykonująca obliczenia. *Computation Resource* to maszyna z zainstalowanym klientem BalticLSC, której zasoby będą wykorzystywane do prowadzenia obliczeń rozproszonych. Oprócz informacji o maszynie, diagram zawiera również dane o klastrze obliczeniowym *Computation Cluster*, który wchodzi w skład *Computation Resource*. Informacje obecne na diagramie (mniej istotne w kontekście niniejszej magisterskiej) to m.in *Computation Machine* - informacje o zasobach maszyny takich jak dostępny procesor lub pamięć, *Cluster Manifest*, w którym zapisane są informacje o sieci oraz *Batch Execution* - posiadający informacje nt. obecnych obliczeń prowadzonych przez maszynę.

Rysunek 3.7 przedstawia słownik *Computation Execution* - wykonanie

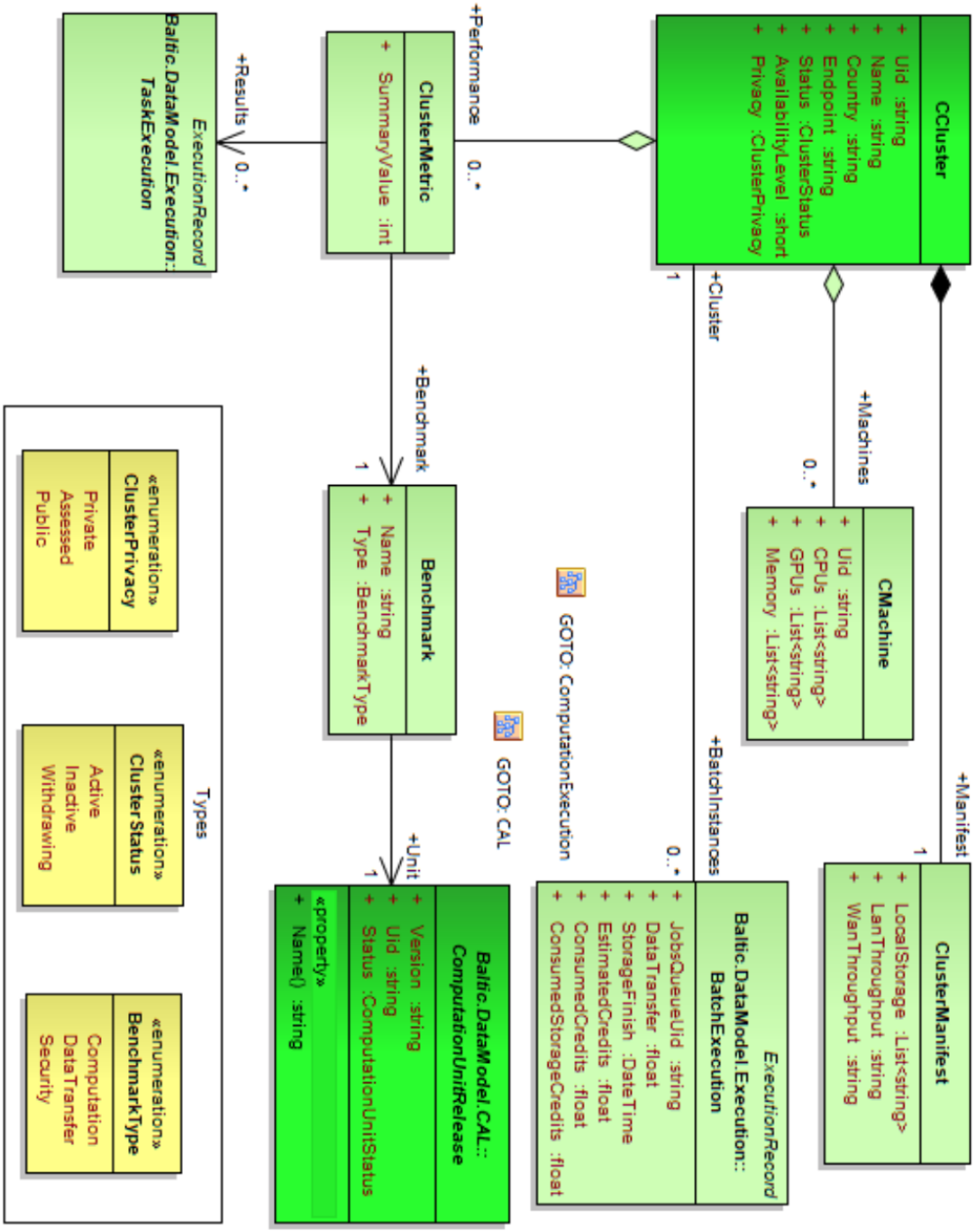


Rysunek 3.5: Fragment słownika *Batch Manager*

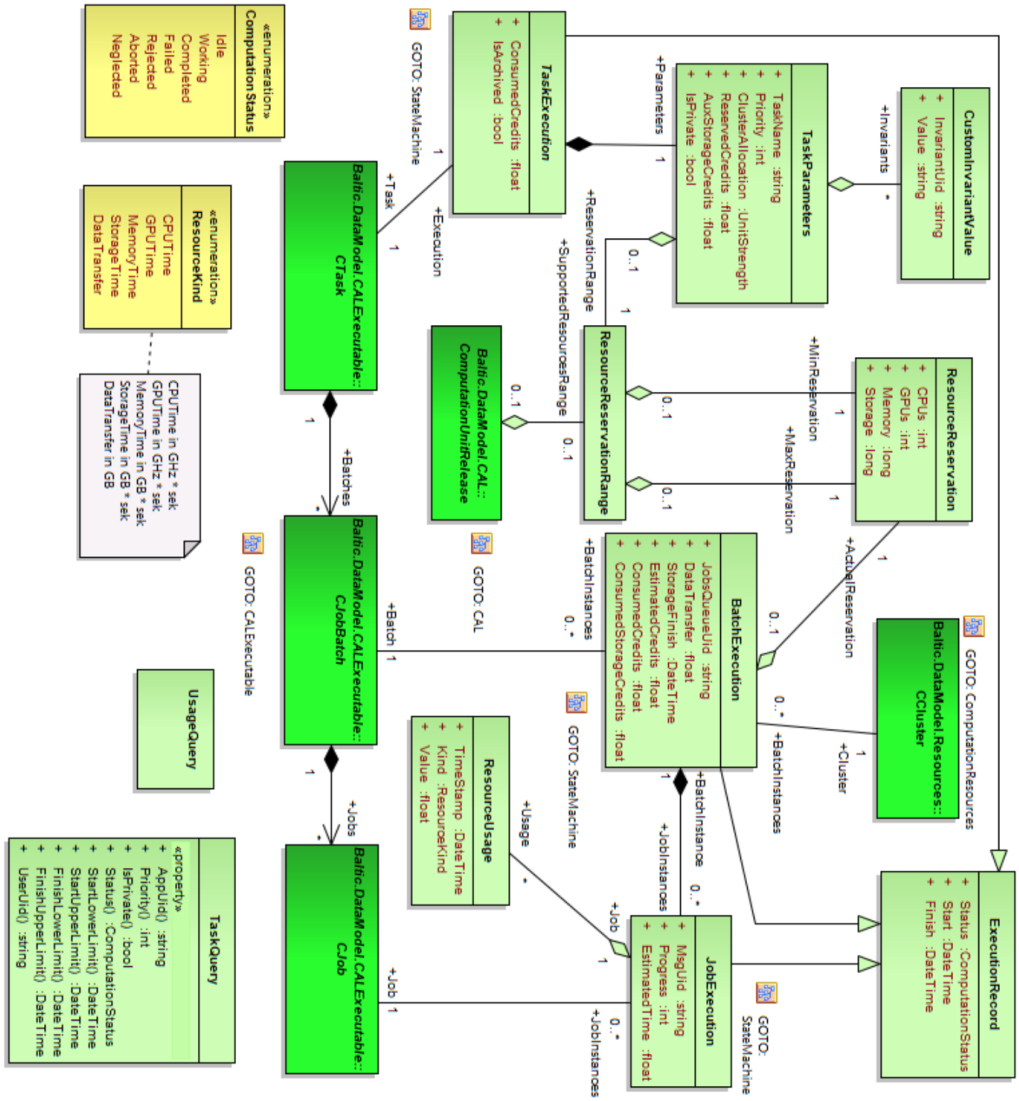
obliczeń zlecane przez *Master Node Backend* dla *Cluster Node*. *Calculcation Exectuable* jest składającym się z kilku kroków zadaniem obliczeniowym, którego wizję przedstawio na rysunku 3.8. Podstawowym zadaniem obliczeniowym jest *Task Execution*. Podzielono go na wiele *Calculation Job Batch*:

- *Calculation Job Batch* opisany przez *Batch Message* definiuje środowisko dla uruchamiania zadań obliczeniowych na klastrze. Jest on podzielony na wiele *Calculation Job*.
- *Calculation Job* opisany przez *Job Message* definiuje pojedyncze zadanie w środowisku konkretnego *Calaculation Job Batch*. Składa się on z wielu *Calculation Data Token*.
- *Calculation Data Token* opisany przez *Token Message* zawiera m.in zestaw danych, które podlegają obliczeniom.

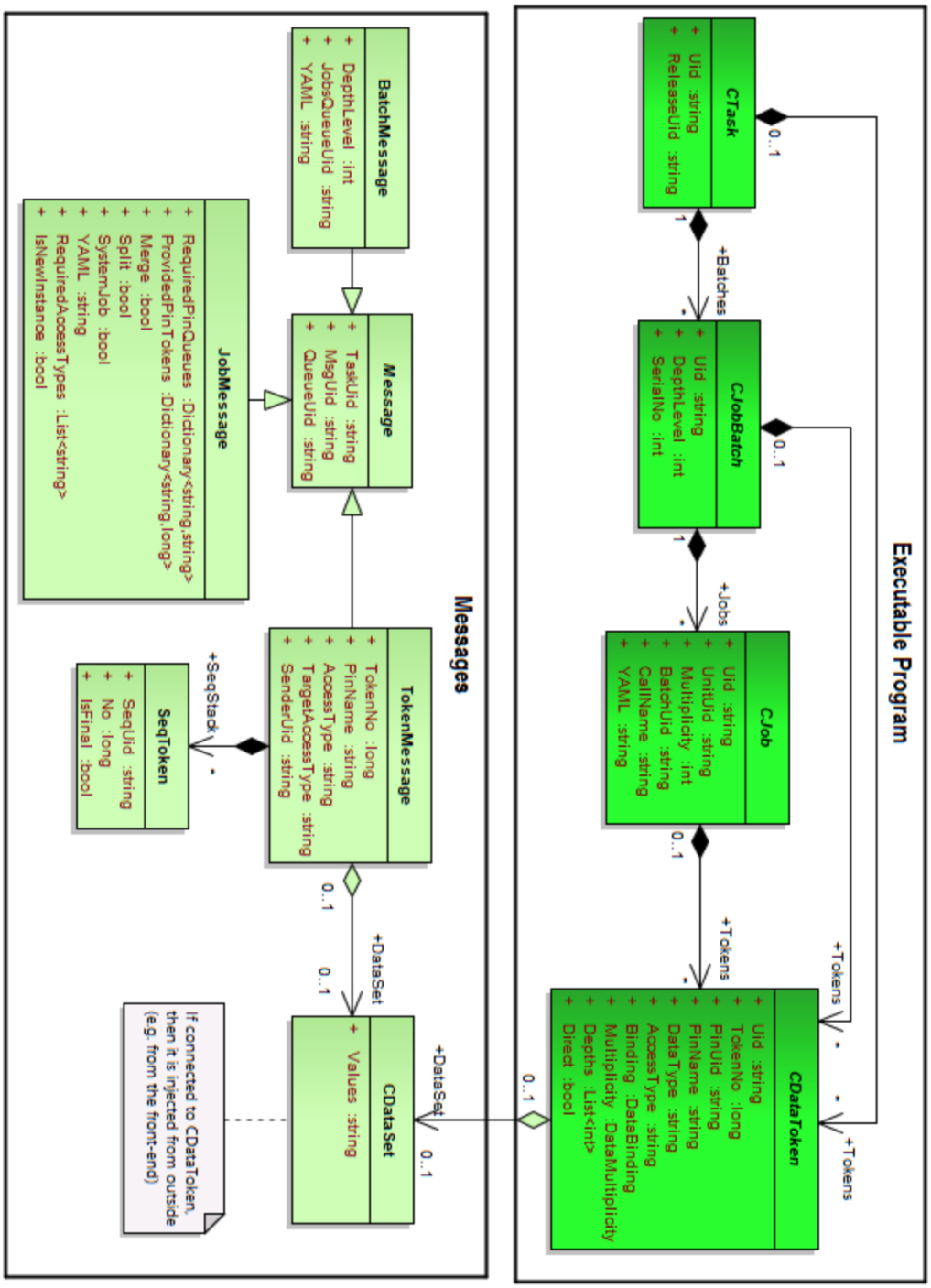
Rysunek 3.7 posiada również definicje mniej istotne w kontekście pracy magisterskiej m.in. *Task Parameters* - informacje o wykonywanym zadaniu obliczeniowym, *Execution Record* - status, czas startu i wykonania zadania oraz *Resource Reservation* - informacje o zasobach maszyny takich jak procesor czy pamięć.



Rysunek 3.6: Słownik *Computation Resource* [37]



Rysunek 3.7: Słownik *Computation Execution* [37]



Rysunek 3.8: Słownik *Calculation Executable* [37]

Rozdział 4

Projekt i implementacja

4.0.1 Weryfikacja obliczeń rozproszonych

W trakcie przeprowadzania obliczeń przez *Computational Cluster* przetwarzane są wiadomości od *Batch Manager* - podmiotu zlecającego. Jak już wspomniano w poprzednim rozdziale, wiadomości te składają z trzech typów:

- *Batch Message* - definiuje środowisko uruchomieniowe
- *Job Message* - definiuje pojedyncze zadanie
- *Token Message* - zawiera m.in zestaw danych, który podlega obliczeniom

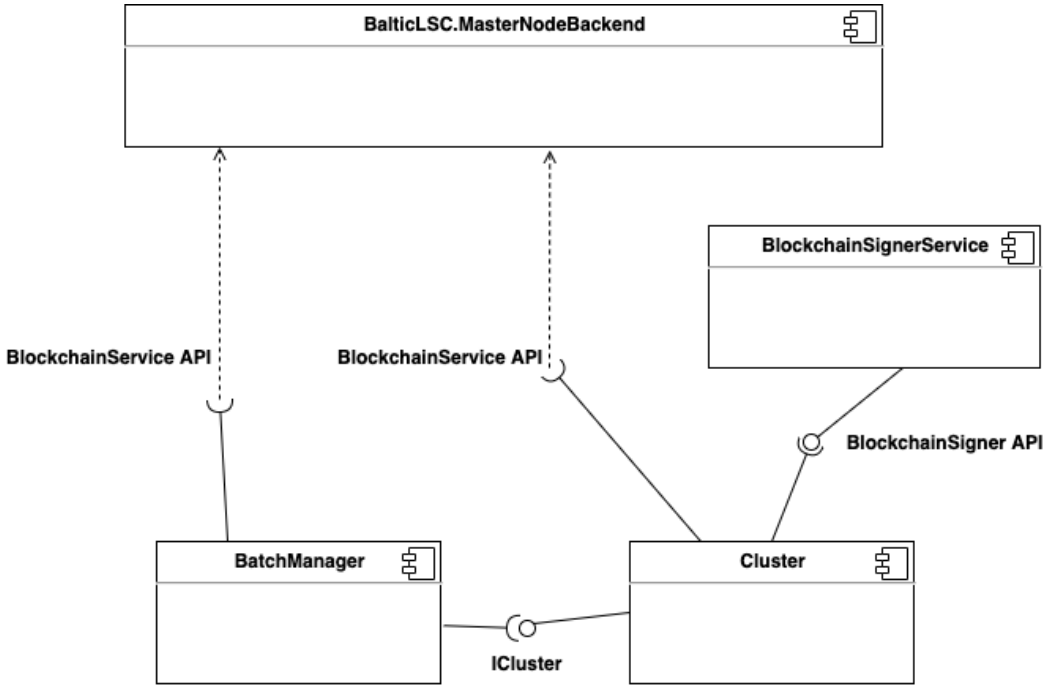
Klaster obliczeniowy po zakończeniu pracy nad poprzednim będzie przetwarzał następny *Token Message*.

W *Token Message* zawarte jest potwierdzenie, że wykonano obliczenia oraz informacja, gdzie znajdują się wytworzone w obliczeniach dane. Dla użytkownika systemu ważne jest, aby mieć pewność, że dane wyprodukowane przez *Computation Cluster* zostały prawidłowo policzone.

Aby spełnić powyższe warunki rozwiązaniem może być zastosowanie technologii Blockchain. Pozwala ona na przeprowadzenie weryfikacji, czy dany obliczenia rozproszone zostały wykonane przez *Computation Cluster*. BalticLSC mógłby posiadać sieć Blockchain zawierającą informację o tym, który *Computation Cluster* dokonał obliczeń. Dzięki posiadanemu dostępowi do identyfikacji wiadomości składających się na zestaw obliczeń możliwe jest dokonanie walidacji. Diagramy zostały wykonane w oprogramowaniu Draw.io [18] w zgodzie ze standardem UML 2.0 [1].

Rysunek 4.1 przedstawia fragment diagramu komponentów *Cluster Node*.

Dodano do niego serwis *Blockchain signer*. *Blockchain signer* pozwala na lokalne podpisywanie transakcji za pomocą prywatnego klucza. W tym celu



Rysunek 4.1: Fragment diagramu komponentów Cluster Node z zastosowaniem Blockchain

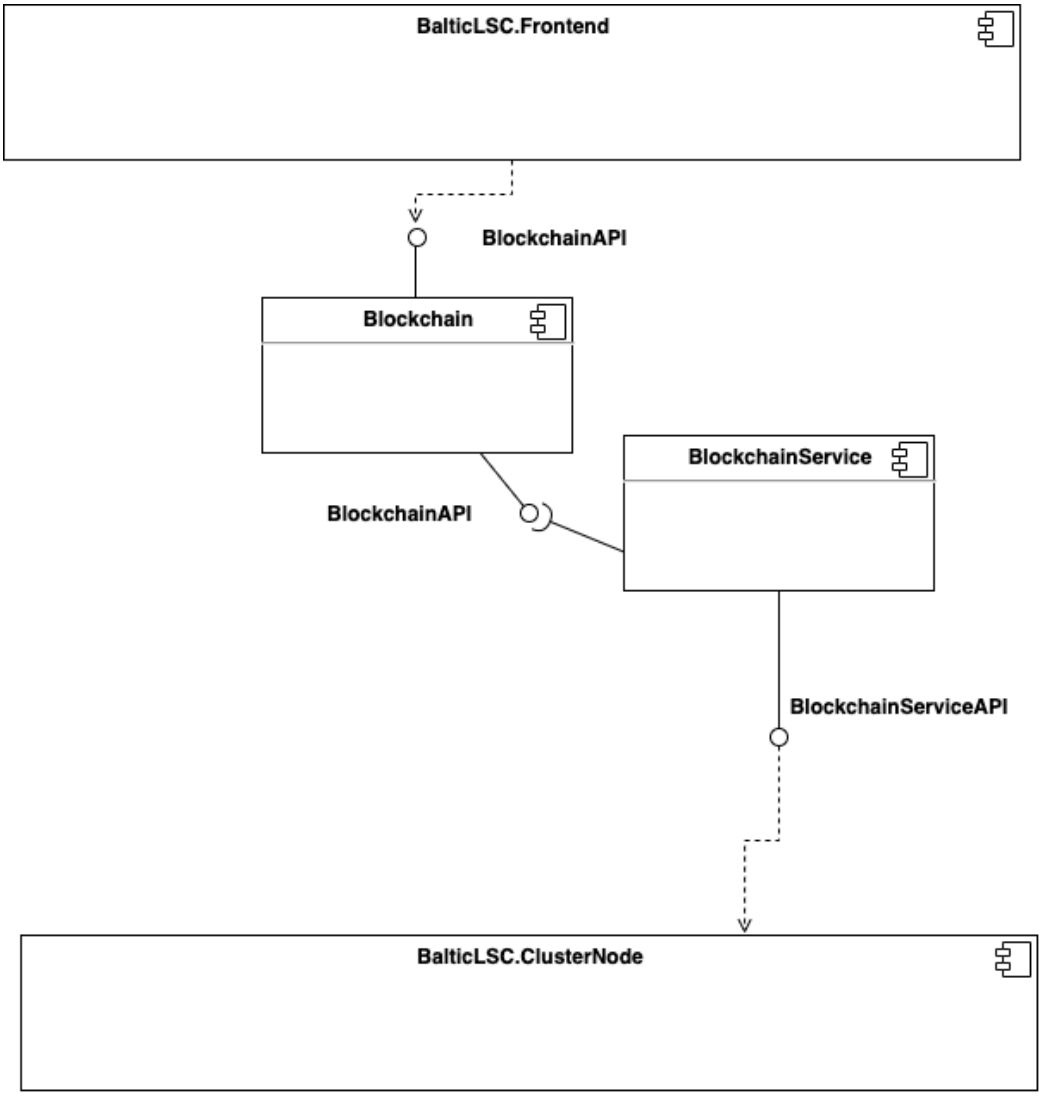
nie ma konieczności połączenia z siecią Blockchain. *Blockchain signer* używany jest przez *Computation Cluster* do stworzenia transakcji zawierającej informację kto wykonał obliczenia.

Rysunek 4.2 przedstawia fragment diagramu komponentów *Master Node Backend*.

Dodano do niego moduł *Blockchain service*, który odpowiada za interakcję BalticLSC z siecią Blockchain. Wszystkie transakcje przekazywane są na sieć Blockchain poprzez *Blockchain service*. Za jego pomocą *Computation Cluster* nadaje transakcję weryfikującą do Blockchain. Jest on także wykorzystywany przez *Batch Manager*, który może dokonywać weryfikacji, czy obliczenia zostały wykonane przez dany *Computation Cluster*.

Do diagramu *Master Node Backend* został również dodany komponent *Blockchain*. Zawiera on w sobie węzeł sieci *Blockchain* oraz przechowuje informacje o weryfikacjach. Jest on używany przez *Blockchain service* oraz *Blockchain viewer* jako bezpośredni punkt dostępowy do Blockchain. Na rysunku 4.3 przedstawiono fragment diagramu komponentów *Frontend*.

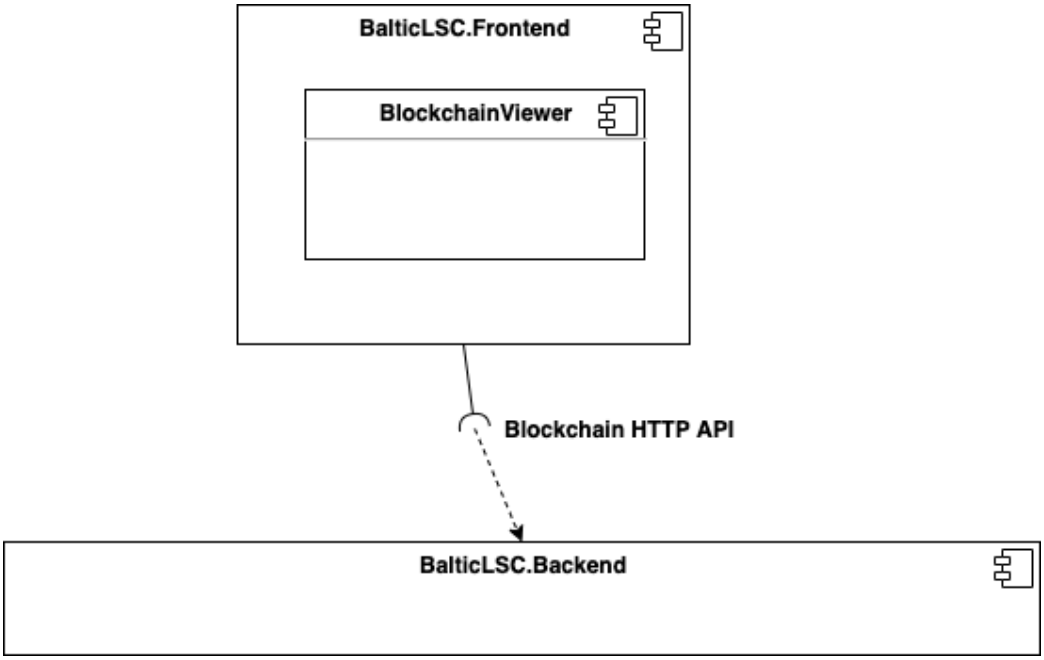
Dodano do niego serwis *Blockchain Viewer*, który pozwala on na to, aby użytkownik końcowy mógł zweryfikować obliczenia rozproszone. Łączy się on



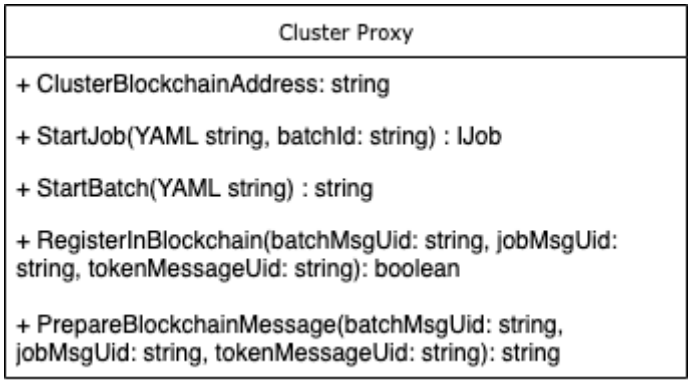
Rysunek 4.2: Fragment diagramu komponentów Master Node Backend z zastosowaniem Blockchain

bezpośrednio z siecią Blockchain poprzez component *Blockchain*. Rysunek 4.4 przedstawia fragmenty diagramu słownika *Cluster Proxy*.

Do *Cluster Proxy* dodano atrybut *ClusterBlockchainAddress* - atrybut typu string definiujący adres Blockchain identyfikujący *Computation Cluster*. Informacje wysłane do sieci Blockchain są podpisane przez klucz prywatny *Computation Cluster* i identyfikowane przez jego adres. Poza powyższymi do *Cluster Proxy* została dodana funkcja *PrepareBlockchainMessage*.

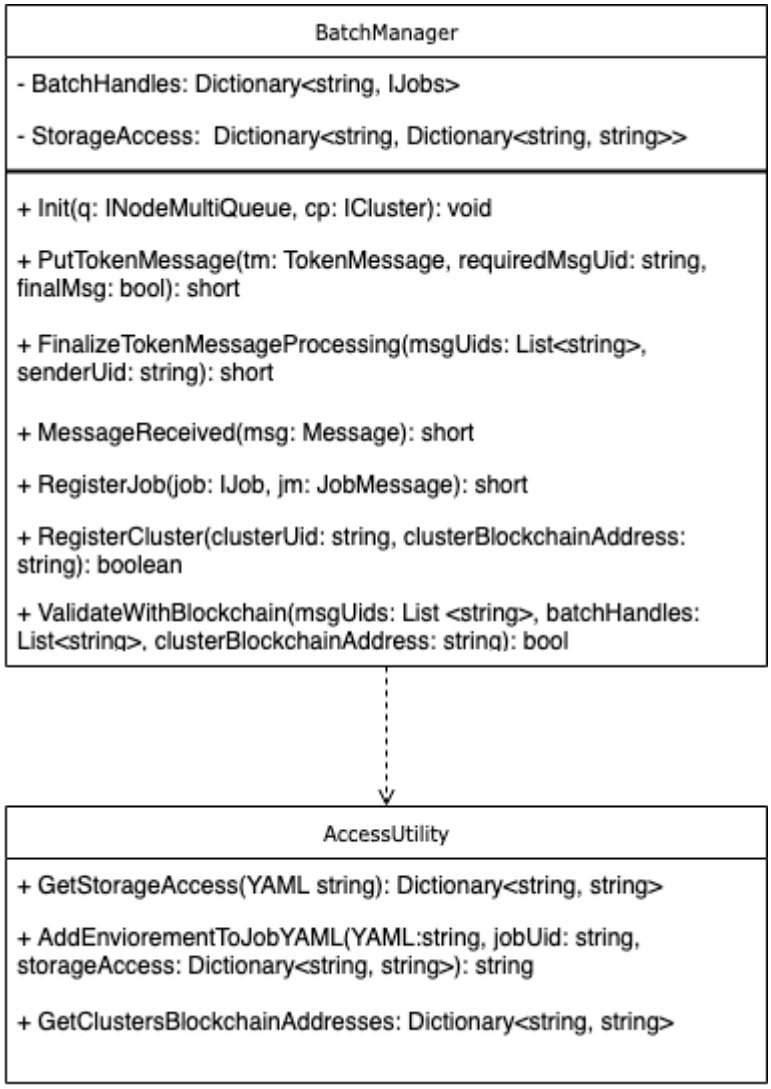


Rysunek 4.3: Fragment diagramu komponentów Frontend Components z zastosowaniem Blockchain



Rysunek 4.4: Fragment słownika *Cluster Proxy* z zastosowaniem Blockchain

Pozwala ona za pomocą serwisu *Blockchain signer* stworzyć podpisaną wiadomość gotową do nadania do Blockchain. Do *Cluster Proxy* została dodana także funkcja *RegisterInBlockchain*. Za pomocą stworzonego serwisu *Blockchain service* pozwala ona na nadanie podpisanej wiadomości do łańcucha Blockchain, kiedy *Computational Cluster* chciałby zarejestrować fakt wyko-

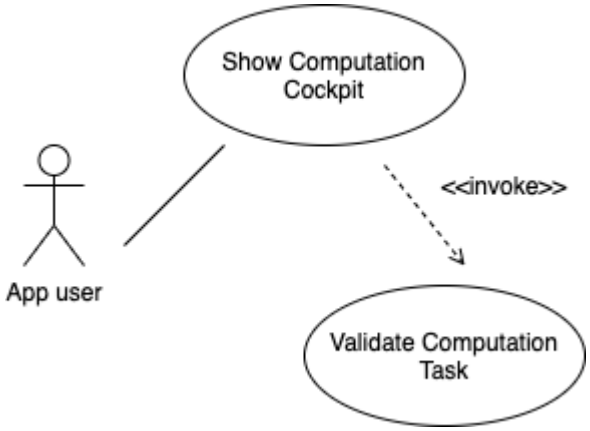


Rysunek 4.5: Fragment słownika *Batch Manager* z zastosowaniem Blockchain

nania obliczeń na Blockchain.

Rysunek 4.5 przedstawia fragment diagramu słownika *Batch Manager*.

Zostały do niego dodane dwie funkcje. Pierwsza z nich to *ValidateWith-Blockchain* na klasie *BatchManager*. Jest ona używana do walidacji obliczeń - pozwala na pozyskanie adresu *Computaton Cluster* - autora obliczeń. Jej parametrami są identyfikatory wiadomości *Batch Message*, *Job Message* oraz *Token Message*. Drugą funkcją jest *GetClustersBlockchainAddress* dodana do klasy *Access Utility*. *Access Utility* przechowuje informację o adresach *Com-*



Rysunek 4.6: Fragment diagram użycia Computation Application Usage

putaton Cluster, a także ułatwia *Batch Manager* odpytywanie o nie.

Rysunek 4.6 przedstawia fragment diagramu przypadków użycia *Computation Application Usage*. Został do niego dodany przypadek zastosowania *Validate Computational Task*.

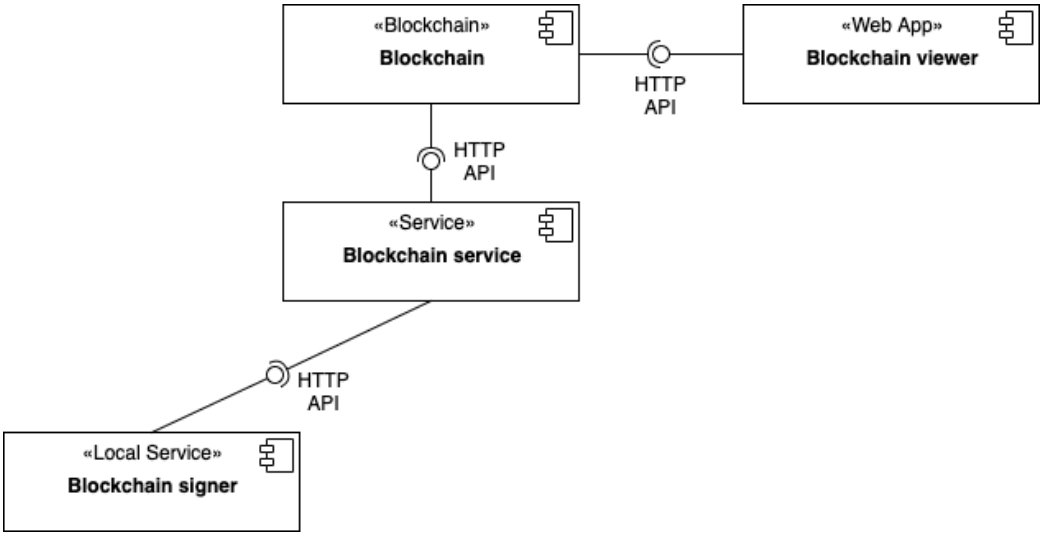
Oprócz automatycznej weryfikacji obliczeń przez *Batch Manager* użytkownik końcowy może zweryfikować dane obliczenia za pomocą dostarczonego interfejsu użytkownika. Po wprowadzeniu identyfikacji wiadomości *Batch Message*, *Job Message* oraz *Token Message* użytkownikowi zwracane jest okno opisujące adres autora obliczeń (*Computation Cluster*).

4.1 Wstęp do projektu

Celem projektu jest próba implementacji walidacji obliczeń rozproszonych za pomocą technologii Blockchain. Pozwoli to uzyskać odpowiedź na pytanie, czy technologia Blockchain jest odpowiednia do tego typu zastosowań i czy można jej użyć w produkcyjnym rozwiązaniu Baltic LSC.

Projekt magisterski składa się z następujących komponentów:

- Modułu *Blockchain*, na który składa się Blockchain jak oraz Smart Contract
- Modułu *Blockchain service* - mikroserwisu służącego do interakcji z siecią Blockchain
- Modułu *Blockchain signer* - mikroserwisu do podpisywania transakcji
- Modułu *Blockchain viewer* - interfejsu użytkownika pozwalającego na weryfikację obliczeń rozproszonych przez użytkownika



Rysunek 4.7: Diagram komponentów projektu

Architektura systemu widoczna jest na diagramie komponentów systemu 4.7.

4.2 Użyte technologie

W projekcie wykorzystano technologie, które mają zastosowanie w produkcyjnych rozwiązaniach integrujących się z Blockchain [24].

Jako główny język projektu wybrano TypeScript. Jest to *superset* JavaScriptu rozwijany przez Microsoft [20]. Główną różnicą między TypeScriptem a JavaScriptem jest statyczne typowanie. Typescipt jest coraz bardziej popularny w społeczności osób posługującym się językiem JavaScript [21]. Typescript transpiluje się do JavaScriptu.

W przypadku Backendu użyto Node.js z frameworkiem HTTP Express. Node.js jest środowiskiem uruchomieniowym JavaScript napisanym w C++ [19]. Framework Express jest obecnie najpopularniejszą aplikacją Webową napisaną w Node.js [21].

Dwoma głównymi klientami dla Ethereum są Geth i Parity [22]. Różnią się oni głównie językiem, w którym zostały napisane - Geth został napisany w Go, Parity w Rust. Z uwagi na osobiste doświadczenie zawodowe, jako klienta Ethereum na potrzeby niniejszej pracy wybrano Parity.

Do napisania Smart Contract zastosowano Solidity. Natomiast do zarządzania Smart Contract został wybrany Truffle suite. Solidity jest jednym z najpopularniejszych języków programowania do tworzenia Smart Contract w

Ethereum. Truffle suite pozwala na automatyczne testowanie czy migrację Smart Contract [23].

W celu interakcji Backendu z Blockchain użyto web3.js.

UI został napisany w języku TypeScript z biblioteką React. React jest jednym z popularniejszych rozwiązań do stworzenia interfejsu użytkownika [21]. Pozwala ona na podzielenie kodu aplikacji na komponenty.

Do testów manualnych oraz prezentacji projektu został użyty Postman.

Jednym z warunków projektu była całkowita jego konteneryzacja. Serwisy zostały skonteneryzowane z użyciem technologii Docker [25]. Pozwala ona na uruchamianie obrazów projektu na dowolnej maszynie.

4.2.1 Moduł Blockchain

W projekcie użyto prywatnej sieci Blockchain opartej o konsensus Proof of Authority. *Validator Node* to węzeł Blockchain podpisujący bloki. W konsensusie Proof of Authority, używając algorytm Aura, każdy *Validator Node* ma przydzielony czas systemowy, w którym może tworzyć nowe bloki. Blockchain jest skonfigurowany poprzez tzw. *chain specification* definiujący właściwości łańcucha:

```
{
  "name": "BalticLsc",
  "engine": {
    "authorityRound": {
      "params": {
        "stepDuration": "2",
        "validators": {
          "list": [
            "0x240518f769856df823c046ddd2aca29bc53eb6ce"
          ]
        }
      }
    }
  },
  "params": {
    "gasLimitBoundDivisor": "0x400",
    "maximumExtraDataSize": "0x20",
    "minGasLimit": "0x1388",
    "networkID": "0x2323",
    "eip155Transition": 0,
    "validateChainIdTransition": 0,
    "eip140Transition": 0,
    "eip211Transition": 0,
    "eip214Transition": 0,
    "eip658Transition": 0
  },
  "genesis": {
    "seal": {
      "authorityRound": {
        "step": "0x0",
        "signature": "0x0000000000000000000000000000000000000000000000000
        ↪  0000000000000000000000000000000000000000000000000
        ↪  0000000000000000000000000000000000000000",
      }
```

```
    },
    "difficulty": "0x20000",
    "gasLimit": "0x8000000"
  },
  "accounts": {
    "0x0000000000000000000000000000000000000001": {
      "balance": "1",
      "builtin": {
        "name": "ecrecover",
        "pricing": {
          "linear": {
            "base": 3000,
            "word": 0
          }
        }
      }
    },
    "0x0000000000000000000000000000000000000002": {
      "balance": "1",
      "builtin": {
        "name": "sha256",
        "pricing": {
          "linear": {
            "base": 60,
            "word": 12
          }
        }
      }
    },
    "0x0000000000000000000000000000000000000003": {
      "balance": "1",
      "builtin": {
        "name": "ripemd160",
        "pricing": {
          "linear": {
            "base": 600,
            "word": 120
          }
        }
      }
    },
    "0x0000000000000000000000000000000000000004": {
      "balance": "1",
      "builtin": {
        "name": "identity",
        "pricing": {
          "linear": {
            "base": 15,
            "word": 3
          }
        }
      }
    },
    "0x240518f769856df823c046ddd2aca29bc53eb6ce": {
      "balance": "160693804425899027554196209234116260252220299378279283530137
      ↪  6"
    }
  }
}
```

Dla celów pracy magisterskiej sieć Blockchain składa się z jednego *Validator Node*. *Validator Node* potrzebuje definicji konta. W tym przypadku konto

reprezentuje organizację należącą do BalticLSC np. Politechnikę Warszawską. W przypadku pracy magisterskiej, adres konta to `0x240518f769856df823c046ddd2aca29bc53eb6ce`.

Chain specification zawiera w sobie min.:

- informację, że *Proof of Authority* będzie wykorzystywany w łańcuchu
- *gensis block* - pierwszy blok w sieci [26]
- informację, że bloki będą produkowane co 2 sekundy
- informację że lista kont *Validator Node* jest stała i definiuje adres `0x240518f769856df823c046ddd2aca29bc53eb6ce`

Plik konfiguracyjny węzła wygląda następująco:

```
[parity]
   chain = "spec/balticLsc.json"

   [ui]
   interface = "all"
   force = true

   [rpc]
   hosts = ["all"]
   apis = ["all"]
   interface = "all"
   cors = ["http://remix.ethereum.org", "http://localhost:3002"]  #
   ↪  allow remix and web ui to access this node

   [websockets]
   interface = "all"
   origins = ["all"]

   [account]
   unlock = ["0x240518f769856df823c046ddd2aca29bc53eb6ce"]
   password = ["spec/pwd"]

   [mining]
   engine_signer = "0x240518f769856df823c046ddd2aca29bc53eb6ce"
```

Specyfikuje on takie aspekty jak:

- punkty dostępu do łańcucha (np. JSON-RPC)
- określenie konta *Validator Node* oraz specyfikacja ścieżki do hasła odszyfrowującego klucz prywatny
- ustawienie CORS, pozwalające na dostęp przez Remix czy aplikację z poziomu localhost [27] (Remix jest internetowym edytorem do Smart Contractów)

Komendy do zbudowania i uruchomienia obrazu Docker oraz REST API modułu są dostępne w Dodatku do pracy magisterskiej.

Dla celów pracy magisterskiej napisano Smart Contract *CalculationVerification*, za pomocą którego weryfikowane są obliczenia rozproszone.

Smart Contract *CalculationVerification.sol* prezentuje się następująco:

```
pragma solidity 0.5.0;

contract CalculationVerification {
    address owner = msg.sender;

    mapping(string => address) executors;

    constructor() public payable {}

    function finishCalculations(address calculationExecutor, string
↪   memory batchMessageId, string memory jobMessageId, string memory
↪   tokenMessageId, bytes memory signature) public {
        /// only owner can send service transactions
        require(msg.sender == owner);

        string memory identifier =
↪   string(abi.encodePacked(batchMessageId, jobMessageId,
↪   tokenMessageId));

        /// calculations can be finished only once
        require(executors[identifier] == address(0));

        bytes32 message =
↪   prefixed(keccak256(abi.encodePacked(calculationExecutor,
↪   batchMessageId, jobMessageId, tokenMessageId)));

        /// will fail if arguments do not match or signer is not
↪   correct
        require(recoverSigner(message, signature) ==
↪   calculationExecutor);

        executors[identifier] = calculationExecutor;
    }

    function getCalculationsExecutor(string memory batchMessageId,
↪   string memory jobMessageId, string memory tokenMessageId) public
↪   view returns (address) {
        string memory identifier =
↪   string(abi.encodePacked(batchMessageId, jobMessageId,
↪   tokenMessageId));
        return executors[identifier];
    }

    function splitSignature(bytes memory sig)
    internal
    pure
    returns (uint8 v, bytes32 r, bytes32 s)
    {
        require(sig.length == 65);

        assembly {
        // first 32 bytes, after the length prefix.
            r := mload(add(sig, 32))
        // second 32 bytes.
            s := mload(add(sig, 64))
        // final byte (first byte of the next 32 bytes).
            v := byte(0, mload(add(sig, 96)))
```

```
        }

        return (v, r, s);
    }

    function recoverSigner(bytes32 message, bytes memory sig)
    internal
    pure
    returns (address)
    {
        (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);

        return ecrecover(message, v, r, s);
    }

    function prefixed(bytes32 hash) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed
↪   Message:\n32", hash));
    }
}
```

Jego głównym celem jest weryfikacja podpisanej wiadomości bez konieczności wysyłania jej z konta osoby, która ją podpisała. Użytkownik używa dedykowanej biblioteki do podpisania wiadomości (np. web3.js). W tym przypadku wiadomość składa się z informacji jaki *Computation Cluster* wykonał dane obliczenia. Wykonane obliczenia są definiowane za pomocą identyfikatorów poszczególnych wiadomości - *Batch Message*, *Job Message* oraz *Token Message*. *Computation Cluster* jest identyfikowany za pomocą adresu. Następnie wiadomość jest podpisana i poddana funkcji skrótu. *Computation Cluster* wykorzystuje w tym celu moduł *Blockchain signer*. Następnie, podpisana wiadomość zostaje wysłana przez *Computation Cluster* do modułu *Blockchain service*. Ostatecznie moduł *Blockchain service* wywołuje Smart Contract z podpisaną wiadomością oraz z określonymi parametrami tj. określenie, który *Computation Cluster* wykonał dane obliczenia oraz identyfikację wiadomości obliczeniowych. Moduł *Blockchain service* wykonuje te czynności przez konto administracyjne, które jako jedyne jest uprawnione do nadawania transakcji Blockchain.

Funkcje Smart Contract działają w następujący sposób:

Funkcja finishCalculations

- Funkcja *ecrecover* użyta w funkcji finishCalculations umożliwia odtworzenie adresu podmiotu podpisującego wiadomość. Jest to możliwe tylko wtedy, kiedy *Computation Cluster* oraz identyfikatory *Batch Message*, *Job Message* oraz *Token Message* podane w podpisanej wiadomości są takie same jak te, które zostały podane w parametrach wywołanej funkcji. Logika kontraktu nie pozwoli na wykonanie transakcji, w której

adres podmiotu wykonującego obliczenia jest inny niż adres *Computation Cluster*, który podpisał wiadomość.

- Jeżeli wszystkie informacje się zgadzają, adres *calculationExecutor* identyfikujący dany *Computation Cluster* jest ustawiany jako wykonawca obliczeń.

Funkcja getCalculationsExecutor

- Pozwala na uzyskanie adresu reprezentującego *Computation Cluster* po podaniu identyfikatorów wiadomości definujących obliczenia tj. *Batch Message*, *Job Message* oraz *Token Message*.
- Jeżeli nikt nie wykonał jeszcze danych obliczeń, zostaną zwrócone same zera czyli pusty adres wykonawcy.

Inicjalizacja Smart Contract:

- zachodzi poprzez wykorzystanie biblioteki Truffle (w projekcie w obrazie Docker)
- konieczne jest, aby węzeł Blockchain nasłuchiwał na połączenia protokołu JSON-RPC

Komendy do zbudowania i uruchomienia obrazu Docker oraz REST API modułu są dostępne w Dodatku do pracy magisterskiej.

4.2.2 Moduł Blockchain signer

Moduł *Blockchain signer* jest skonteneryzowaną przez Docker aplikacją napisaną w Node.js. Wykorzystuje on framework Express do serwera HTTP używającego REST. HTTP i REST są sposobem połączenia mikro serwisów w BalticLSC. Głównym celem *Blockchain signer* jest podpisywanie wiadomości przez *Computation Cluster* bez konieczności połączenia do węzła Blockchain.

Docelowo moduł ten miałby się znajdować na *Computation Resource*. Wykonując dane obliczenia rozproszone identyfikowane przez wiadomości *Batch Message*, *Job Message* oraz *Token Message*, *Computation Cluster* podpisuje wiadomość. Składa się ona z informacji na temat wiadomości obliczeniowych oraz identyfikujących *Computation Cluster*, który wykonał dane obliczenia. Podpisywanie jest możliwe dzięki kluczowi prywatnemu znajdującemu się w zmiennych środowiskowych przy uruchomieniu *Blockchain signer*.

Głowna logika działania aplikacji zawarta jest w serwisie *BlockchainSignerService*:

```
import Web3 from 'web3'
import {Mixed} from 'web3-utils'
import {Sign, Account} from 'web3-eth-accounts'

const web3 = new Web3('http://localhost:8545')

export class BlockchainSignerService {
    private web3: Web3
    private signerAccount: Account

    constructor() {
        this.web3 = web3
        this.signerAccount = this.web3.eth.accounts.
        ↪  privateKeyToAccount(process.env.SIGNER_PRIVATE_KEY)
    }

    async signComputations(batchMessageId: string, jobMessageId:
    ↪  string, tokenMessageId: string): Promise<Sign> {
        const signedTransaction = this.signTransaction({type:
        ↪  "address", value: this.signerAccount.address}, {type:
        ↪  "string", value: batchMessageId},
            {type: "string", value: jobMessageId}, {type: "string",
            ↪  value: tokenMessageId})
        return signedTransaction
    }

    private async signTransaction(...blockchainParams: Mixed[]):
    ↪  Promise<Sign> {
        const hash = this.web3.utils.soliditySha3(...blockchainParams)
        const signedMessage = await this.web3.eth.accounts.sign(hash,
        ↪  this.signerAccount.privateKey)
        return signedMessage
    }
}
```

W momencie startowania modułu *Blockchain signer* podawany jest klucz prywatny w zmiennych środowiskowych reprezentujący dany *Computation Cluster*. Adres Blockchain identyfikujący *Computation Cluster* będzie zawsze taki sam dla danego klucza prywatnego.

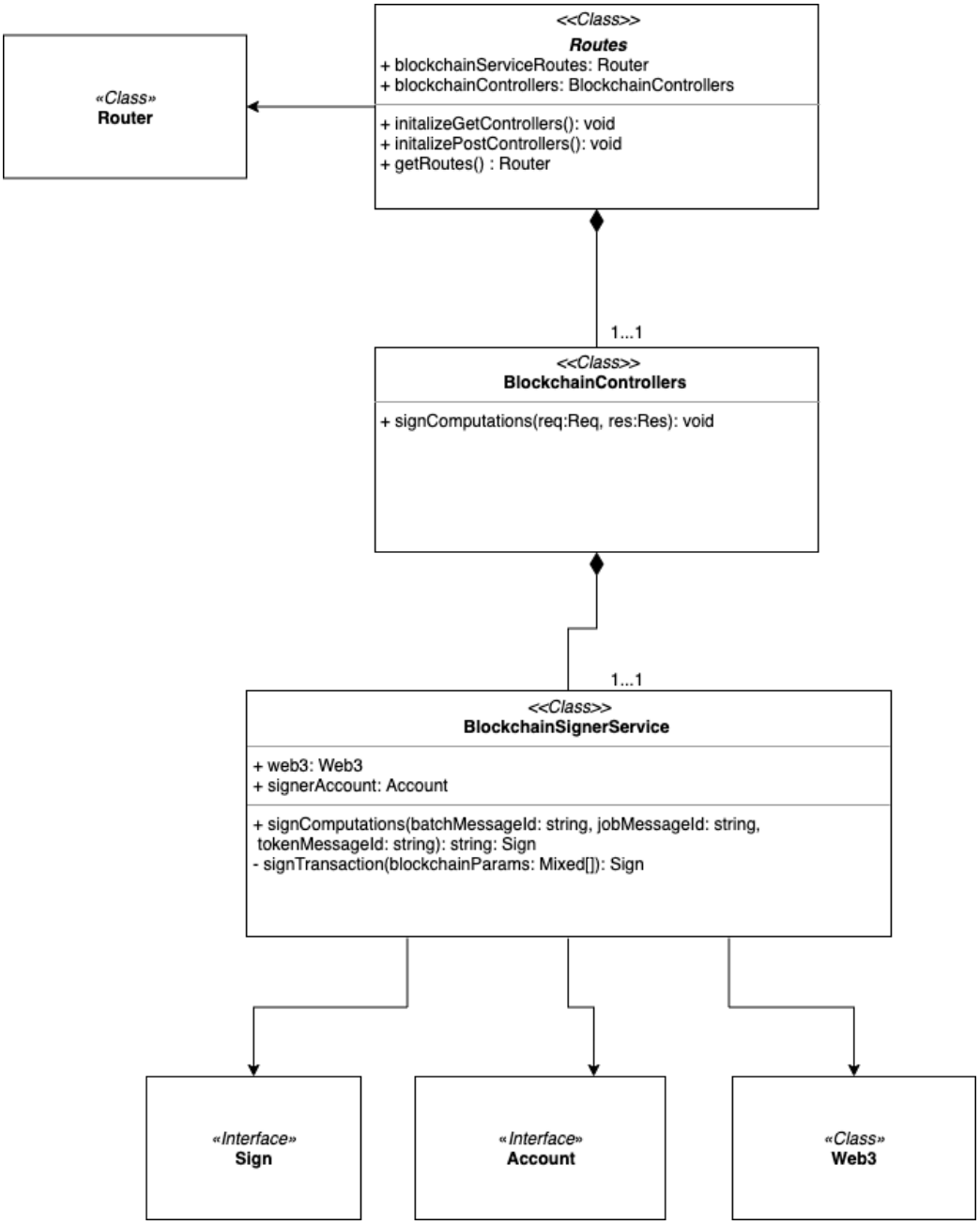
Funkcja *signComputations* modułu *Blockchain signer* wykorzystuje bibliotekę web3.js do podpisania transakcji składającej się z identyfikatorów wiadomości definiujących dane obliczenia. Reszta modułu *Blockchain signer* to serwer HTTP z odpowiednimi kontrolerami.

Komendy do zbudowania i uruchomienia obrazu Docker oraz REST API modułu są dostępne w Dodatku do pracy magisterskiej.

Diagram modułu *Blockchain signer* jest dostępny na rysunku 4.8.

4.2.3 Moduł Blockchain service

Moduł *Blockchain service* jest skonteneryzowaną przez Docker aplikacją napisaną w Node.js. Wykorzystuje on framework Express do serwera HTTP



Rysunek 4.8: Diagram klas modułu *Blockchain signer*

korzystającego z REST API. HTTP i REST są sposobem połączenia mikro serwisów w BalticLSC.

Głównym celem modułu *Blockchain service* jest nadawanie transakcji do

sieci Blockchain oraz weryfikacja obliczeń przez Blockchain. Jest on wykorzystywany przez *Computation Cluster* do zarejestrowania obliczeń na Blockchain. Drugim z zastosowań jest użycie przez *Batch Manager* do zweryfikowania obliczeń po ich wykonaniu przez *Computation Cluster*.

Moduł *Blockchain service* wykonuje te czynności poprzez konto administracyjne, które jako jedyne jest uprawnione do nadawania transakcji Blockchain.

Główna logika działania aplikacji zawarta jest w serwisie *ComputationsVerificationBlockchainService*

```
import Web3 from 'web3'
import {Account} from 'web3-eth-accounts'
import {Contract} from 'web3-eth-contract'
import {PromiEvent} from 'web3-eth'
import {contract} from '../assets/smartContract'

const web3 = new Web3(process.env.BLOCKCHAIN_NODE_URL ||
↪  "http://localhost:8545")

export class ComputationsVerificationBlockchainService {
    private web3: Web3
    private serviceAccount: Account
    private calculationVerificationContract: Contract

    constructor() {
        this.web3 = web3
        this.serviceAccount =
        ↪  this.web3.eth.accounts.privateKeyToAccount
        ↪  (process.env.SERVICE_ACCOUNT_PRIVATE_KEY)
        this.calculationVerificationContract = new
        ↪  this.web3.eth.Contract(contract.abi as any,
        ↪  process.env.CONTRACT_ADDRESS)
    }

    async getComputationsAuthor(batchMessageId: string, jobMessageId:
    ↪  string, tokenMessageId: string): Promise<string> {
        const authorAddress = await
        ↪  this.calculationVerificationContract.methods.
        getCalculationsExecutor(batchMessageId, jobMessageId,
        ↪  tokenMessageId).call({from: this.serviceAccount.address})
        return authorAddress
    }

    private async getNextNonce(address: string): Promise<number> {
        const count = await this.web3.eth.getTransactionCount(address)
        return count
    }

    async submitTransaction(author: string, batchMessageId: string,
    ↪  jobMessageId: string, tokenMessageId: string,
    ↪  signedTransaction: string): PromiEvent {
        const contractCallData =
        ↪  this.calculationVerificationContract.methods.
        ↪  finishCalculations(author, batchMessageId, jobMessageId,
        ↪  tokenMessageId, signedTransaction).encodeABI()
```

```
        const gasEstimate = await
        ↪   this.calculationVerificationContract.methods.
        ↪   finishCalculations(author, batchMessageId, jobMessageId,
        ↪   tokenMessageId, signedTransaction).estimateGas({from:
        ↪   this.serviceAccount.address})
        const nextNonce = await
        ↪   this.getNextNonce(this.serviceAccount.address)
        const txData = {
            from: this.serviceAccount.address,
            to: this.calculationVerificationContract. options.address,
            data: contractCallData,
            nonce: nextNonce,
            gas: gasEstimate
        }
        console.log("Tx data to send", txData)
        const signedTx = await
        ↪   this.web3.eth.accounts.signTransaction(txData,
        ↪   this.serviceAccount.privateKey)
        const receipt = await this.web3.eth.
        ↪   sendSignedTransaction(signedTx.rawTransaction)
        return receipt
    }
}
```

Serwis ten łączy się z węzłem Blockchain poprzez bibliotekę web3.js poprzez wykorzystanie protokołu JSON-RPC.

Funkcje *ComputationsVerificationBlockchainService* działają w następujący sposób:

Funkcja submitTransaction

- Służy do wysyłania transakcji na Blockchain.
- Składa się ona z adresu *Computation Cluster*, identyfikacji wiadomości opisujących dane obliczenia oraz wiadomości podpisanej przez *Computation Cluster*.
- Transakcja jest podpisana przez konto administracyjne.
- Po wysłanej transakcji serwis oczekuje na potwierdzenie włączenia transakcji w łańcuch bloków.

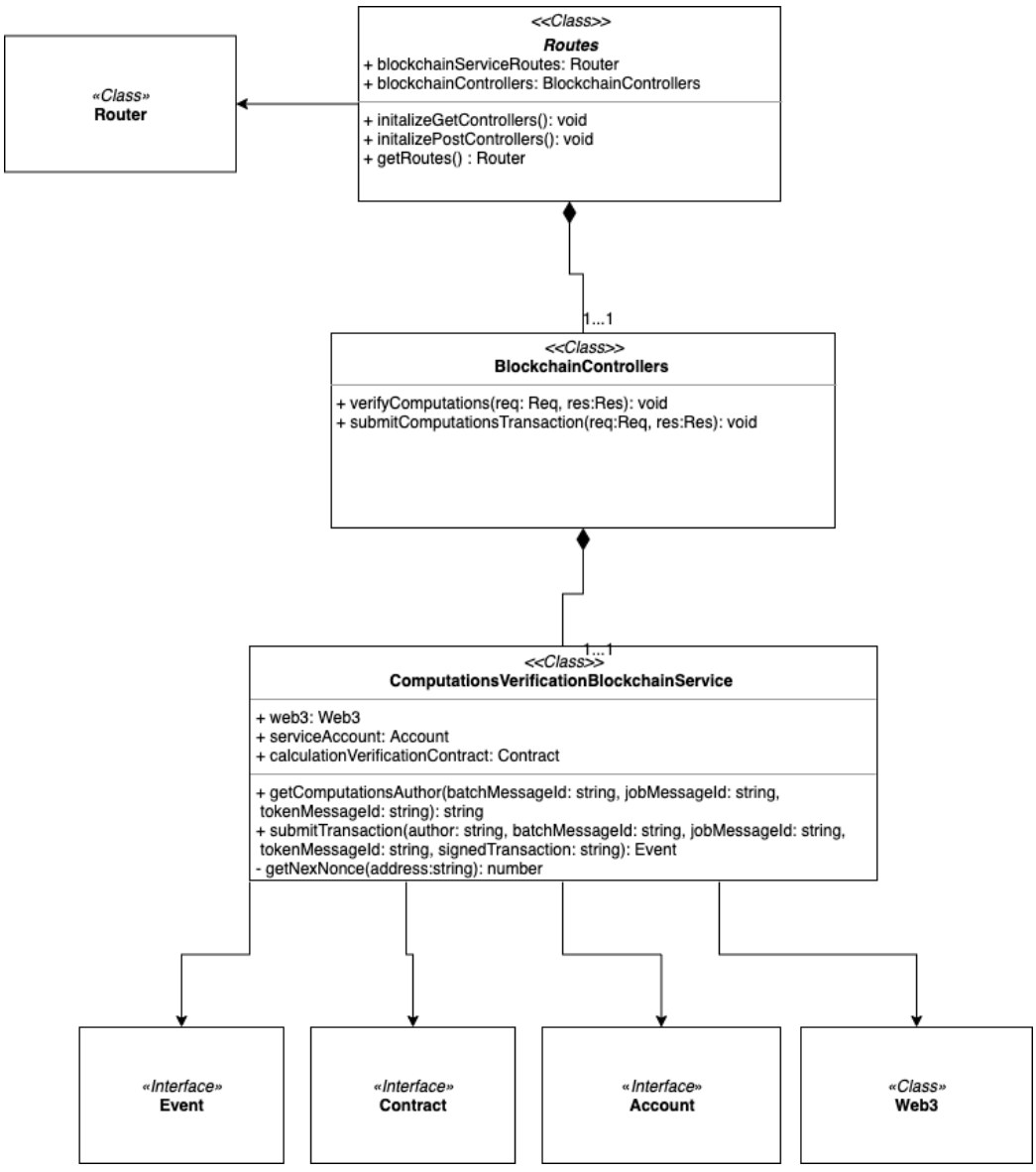
Funkcja getComputationsAuthor

- Zwraca ona adres identyfikujący *Computation Cluster*, który przeprowadził obliczenia opisane wiadomościami *Batch Message*, *Job Message* oraz *Token Message*.
- Jeżeli adres składa się z samych 0 świadczy to o tym, że obliczenia nie zostały jeszcze wykonane.

- Po wysłaniu transakcji serwis oczekuje na potwierdzenie włączenia jej w łańcuch bloków.

Komendy do zbudowania i uruchomienia obrazu Docker oraz REST API modułu są dostępne w Dodatku do pracy magisterskiej.

Diagram modułu *Blockchain service* przedstawiono na rysunku 4.9.



Rysunek 4.9: Diagram klas modułu *Blockchain service*

4.2.4 Moduł Blockchain viewer

Moduł *Blockchain viewer* jest interfejsem użytkownika pozwalającym zweryfikować dane obliczenia opisane wiadomościami *Batch Message*, *Job Message* oraz *Token Message*.

Moduł *Blockchain viewer* łączy się bezpośrednio do sieci Blockchain, pozwalając użytkownikowi zweryfikować obliczenia tylko i wyłącznie w oparciu o sieć Blockchain. Może to służyć w przyszłości do celów użytkowych lub np. audytowych.

Po wprowadzeniu identyfikacji danych wiadomości *Batch Message*, *Job Message* oraz *Token Message* interfejs użytkownika połączy się z siecią Blockchain i zwróci adres wykonawcy obliczeń. Adres identyfikuje *Computation Cluster*.

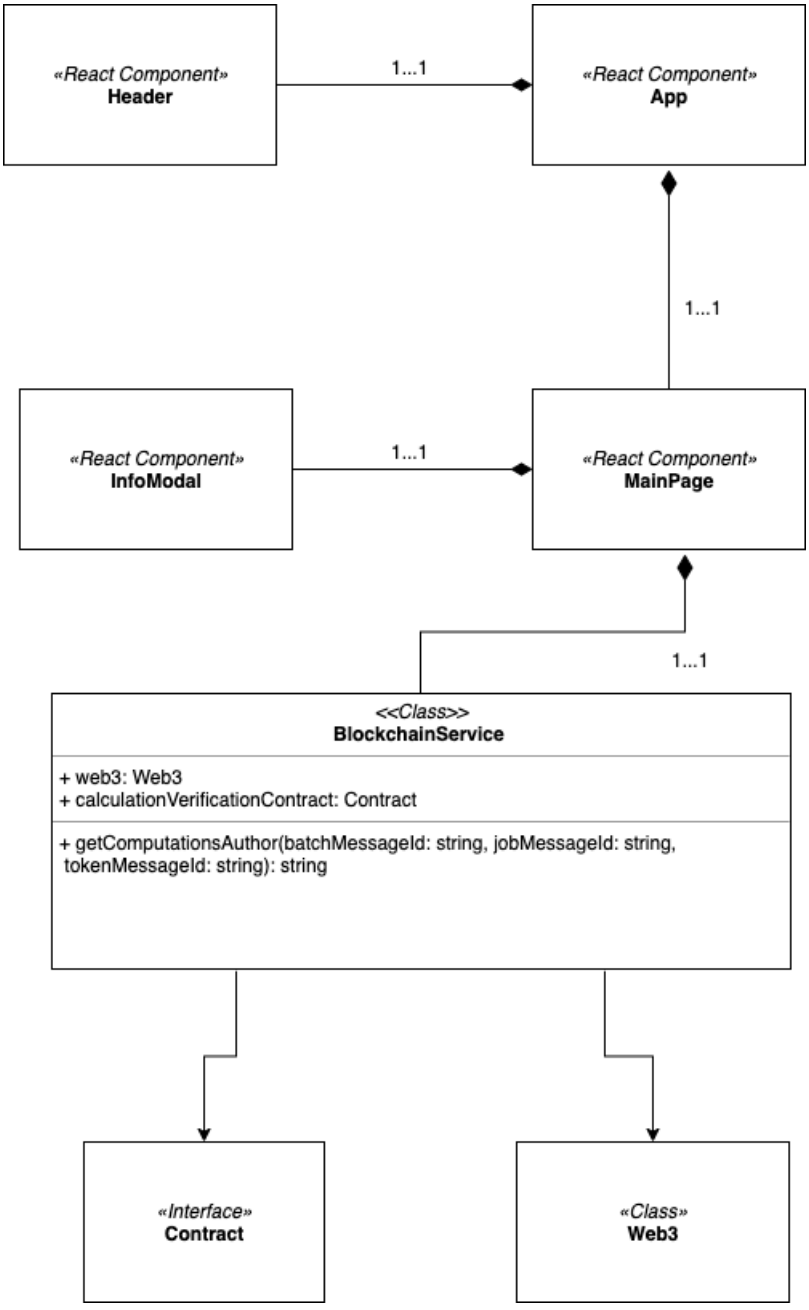
Klasa dokonująca interakcję z Blockchainem czyli *Blockchain Service* wygląda następująco:

```
import Web3 from 'web3'
import {contract} from './contract'
import Contract from 'web3/eth/contract'

const web3 = new Web3(process.env.REACT_APP_BLOCKCHAIN_HOST ||
↪  "http://localhost:8545")

export class BlockchainService {
    private web3: Web3
    private calculationVerificationContract: Contract

    constructor() {
        this.web3 = web3
        this.calculationVerificationContract = new
        ↪  this.web3.eth.Contract(contract.abi as any,
        ↪  process.env.REACT_APP_CONTRACT_ADDRESS)
    }

    async getComputationsAuthor(batchMessageId: string, jobMessageId:
    ↪  string, tokenMessageId: string): Promise<string> {
        const authorAddress = await
        ↪  this.calculationVerificationContract.methods.
        ↪  getCalculationsExecutor(batchMessageId, jobMessageId,
        ↪  tokenMessageId).call()
        return authorAddress
    }
}
```

Funkcja *getComputationsAuthor* po podaniu identyfikacji wiadomości *Batch Message*, *Job Message* oraz *Token Message* opisujących obliczenia zwraca adres identyfikujący *Computation Cluster* który je wykonał.

Komendy do zbudowania i uruchomienia obrazu Docker oraz REST API modułu są dostępne w Dodatku do pracy magisterskiej.

Diagram modułu *Blockchain viewer* jest dostępny na rysunku 4.10.



Rysunek 4.10: Diagram klas modułu *Blockchain viewer*

Rozdział 5

Studium przypadku

5.0.1 Wstęp

Studium przypadku zaczyna się w momencie wykonania obliczeń rozproszonych przez BalticLSC. Tak jak miało to miejsce w rozdziale opisującym działanie BalticLSC, *Master Node* zleca dla *Cluster Node* wykonanie zadania obliczeniowego. Na *Cluster Node* umieszczone są moduły BalticLSC takie jak *Batch Manager* oraz niezaufany *Computation Cluster* wykonujący obliczenia. *Batch Manager* znajdujący się na *Cluster Node* zleca obliczenia poprzez wiadomości przetwarzane przez *Computation Cluster*. Wysyła on następujące wiadomości.

- *Batch Message* - definiuje środowisko uruchomieniowe
- *Job Message* - definiuje pojedyncze zadanie
- *Token Message* - zawiera m.in zestaw danych, który podlega obliczeniom

Po zrealizowaniu części obliczeń opisywanych przez *Computation Data Token* (element w czasie wykonywania opisany przez *Token Message*), *Computation Cluster* może zapisać ten fakt na Blockchain.

Dla celów pracy magisterskiej (w tym przeprowadzenia pełnego studium przypadku) stworzono zestaw kontenerów uruchamianych automatycznie. Wykorzystana tutaj zostaje technologia Docker Compose [29]. Konternery służą do demonstracji działania systemu oraz do przeprowadzenia pełnego studium przypadku użycia.

5.0.2 Moduł Blockchain

Aby wykorzystywać sieć Blockchain do weryfikacji obliczeń, należy uruchomić węzeł Blockchain oraz zainicjalizować na nim *Smart Contract* od-

powiedzialny za logikę biznesową. Do tego celu wykorzystuje się stworzony plik Docker Compose. Za jego pomocą uruchamia się jeden węzeł Blockchain (Parity) oraz inicjalizuje się na nim *Smart Contract CalculationVerification*. Adres *Smart Contractu CalculationVerification* będzie wykorzystywany przez resztę modułów.

W głównym katalogu projektu należy uruchomić komendę:

```
docker-compose -f docker-compose-blockchain.yaml up
```

Wiadomości odpowiedzialne za uruchomienia węzła Blockchain będą widoczne w logach *blockchain node*. Wiadomości odpowiedzialne za inicjalizację kontraktów będą widoczne w logach *blockchain deployer*.

```
blockchain_node        | Loading config file from spec/config.toml
blockchain_node        | 2019-09-30 09:35:02 UTC Starting
↪  Parity-Ethereum/v2.4.9-stable-
↪  691580c-20190701/x86_64-linux-gnu/rustc1.35.0
blockchain_node        | 2019-09-30 09:35:02 UTC Keys path
↪  /root/.local/share/io.parity.ethereum/keys/BalticLsc
blockchain_node        | 2019-09-30 09:35:02 UTC DB path
↪  /root/.local/share/io.parity.ethereum/
↪  chains/BalticLsc/db/ea4452d084cdd17f
blockchain_node        | 2019-09-30 09:35:02 UTC State DB
↪  configuration: fast
blockchain_node        | 2019-09-30 09:35:02 UTC Operating mode: active
blockchain_node        | 2019-09-30 09:35:02 UTC Configured for
↪  BalticLsc using AuthorityRound engine
blockchain_node        | 2019-09-30 09:35:02 UTC Listening for new
↪  connections on 0.0.0.0:8546.
blockchain_deployer    |
blockchain_deployer    | > smart-contracts@1.0.0 migrate /usr/src/app
blockchain_deployer    | > truffle migrate
blockchain_deployer    |
blockchain_node        | 2019-09-30 09:35:07 UTC Public node URL:
↪  enode://66fe2043492358d07b1d094373c0080e5ef27a504a
↪  99235484e1e5c558f9020c2aa0a7ff6226b9582906943a3ab8a9e044f513
↪  95d41264c5c54c0e19eb380717@172.18.0.2:30303
blockchain_deployer    | Important
blockchain_deployer    | If you're using an HDWalletProvider, it must
↪  be Web3 1.0 enabled or your migration will hang.
blockchain_deployer    |
blockchain_deployer    |
blockchain_deployer    | Starting migrations...
blockchain_deployer    | ======================
blockchain_deployer    | > Network name:    'development'
blockchain_deployer    | > Network id:      8995
blockchain_deployer    | > Block gas limit: 133433221
blockchain_deployer    |
blockchain_deployer    |
blockchain_deployer    | 1_initial_migration.js
blockchain_deployer    | ======================
blockchain_deployer    |
blockchain_deployer    |    Deploying 'Migrations'
blockchain_deployer    |    ----------------------
blockchain_node        | 2019-09-30 09:35:08 UTC Fallback to
↪  `BlockId::Latest`
```

```
blockchain_deployer    |    > transaction hash:
↪  0xf908fdb3e26cfcab9733a9f0f8dc6d9b57af55108a72d4b0040d70cc0bac4349
blockchain_deployer    | - Blocks: 0            Seconds: 0
blockchain_node        | 2019-09-30 09:35:10 UTC Imported #7
↪  0x8006...9f29 (1 txs, 0.28 Mgas, 3 ms, 1.52 KiB)
blockchain_node        | 2019-09-30 09:35:10 UTC Transaction mined
↪  (hash
↪  0xf908fdb3e26cfcab9733a9f0f8dc6d9b57af55108a72d4b0040d70cc0bac4349)
blockchain_deployer    |    > Blocks: 0            Seconds: 0
blockchain_deployer    |    > contract address:
↪  0x1b1041923899aa31A7d9C029b565D68FBcBd6FCD
blockchain_deployer    |    > account:
↪  0x7bb8e0B0b5f7d3B510CAA8CC95196Bbc026A66F7
blockchain_deployer    |    > balance:             0
blockchain_deployer    |    > gas used:            284908
blockchain_deployer    |    > gas price:           0 gwei
blockchain_deployer    |    > value sent:          0 ETH
blockchain_deployer    |    > total cost:          0 ETH
blockchain_deployer    |
blockchain_deployer    |    > Saving artifacts
blockchain_deployer    |    -------------------------------------
blockchain_deployer    |    > Total cost:                  0 ETH
blockchain_deployer    |
blockchain_deployer    |
blockchain_deployer    | 2_deploy_calculation_verification.js
blockchain_deployer    | ====================================
blockchain_deployer    |
blockchain_deployer    |    Replacing 'CalculationVerification'
blockchain_deployer    |    ----------------------------------
blockchain_deployer    |    > transaction hash:
↪  0x5441e709e0e5ec0aae2d45e5996c8b2002db3cdac28dda4728135345ada35557
blockchain_deployer    | - Blocks: 0            Seconds: 0
blockchain_node        | 2019-09-30 09:35:14 UTC Transaction mined
↪  (hash
↪  0x5441e709e0e5ec0aae2d45e5996c8b2002db3cdac28dda4728135345ada35557)
blockchain_node        | 2019-09-30 09:35:14 UTC Imported #8
↪  0xbe9a...474a (1 txs, 0.59 Mgas, 0 ms, 2.62 KiB)
blockchain_deployer    |    > Blocks: 1            Seconds: 3
blockchain_deployer    |    > contract address:
↪  0x6dFb238932440E90D11C5d76E7b894B6268870Ec
blockchain_deployer    |    > account:
↪  0x7bb8e0B0b5f7d3B510CAA8CC95196Bbc026A66F7
blockchain_deployer    |    > balance:             0
blockchain_deployer    |    > gas used:            588894
blockchain_deployer    |    > gas price:           0 gwei
blockchain_deployer    |    > value sent:          0 ETH
blockchain_deployer    |    > total cost:          0 ETH
blockchain_deployer    |
blockchain_deployer    |    > Saving artifacts
blockchain_deployer    |    -------------------------------------
blockchain_deployer    |    > Total cost:                  0 ETH
blockchain_deployer    |
blockchain_deployer    |
blockchain_deployer    | Summary
blockchain_deployer    | =======
blockchain_deployer    | > Total deployments:   2
blockchain_deployer    | > Final cost:          0 ETH
blockchain_deployer    |
blockchain_deployer exited with code 0
```

Docker Compose uruchomił węzeł Blockchain oraz zainicjalizował dwa

Smart Contract. Pierwszy z nich to *Smart Contract Migrations*, który jest kontraktem administracyjnym Truffle.js. Odpowiada on za migrację czyli przechowywanie historii poprzedniej wersji kontraktów. Drugi adres zainicjalizowanego *Smart Contract*-u to *Smart Contract* weryfikacyjny *CalculationVerification*. Jego adres w tym przypadku to `0x6dFb238932440E90D11C5d76E7b894B6268870Ec`.

Wystartowany Docker Compose ma przygotowany jeden węzeł Blockchain oraz *Smart Contract CalculationVerification*.

Adres *Smart Contract CalculationVerification* należy skopiować do pliku Docker Compose *docker-compose-backend*. Dzięki temu moduły *Blockchain viewer*, *Blockchain signer* oraz *Blockchain service* będą posiadały informację na jego temat.

5.0.3 Moduł Blockchain viewer, Blockchain signer oraz Blockchain service

Dla celów studium przypadku zaimplementowano Docker Compose łączący w sobie obrazy *Blockchain viewer*, *Blockchain signer* oraz *Blockchain service*.

Po wykonaniu komendy:

```
docker-compose -f docker-compose-backend.yaml up
```

pojawią się następujące informacje:

```
blockchain_viewer     | INFO: Accepting connections at
↪  http://localhost:3002
blockchain_signer     | App listening on port 3000!
blockchain_service    | App listening on port 3001!
```

Świadczą one o poprawnej inicjalizacji kontenerów *Blockchain viewer*, *Blockchain signer* oraz *Blockchain service*.

Wszystkie serwisy potrzebne do rozpoczęcia studium przypadku zostały wystartowane. Przyjmijmy, że *Computation Cluster* wykonał obliczenia składające się z wiadomości o identyfikatorach:

- *Batch Message* 3a7517ac-1f79-4d1f-aaa3-6232b49993fc
- *Job Message* dfca9a1d-d049-4d7c-88a4-51db9d508610
- *Token Message* 125e9809-0017-4959-ac05-cb197046faa7

Jednym z modułów obecnych na *Computation Resource* jest *Blockchain signer*. Służy on do podpisywania wiadomości przez *Computation Cluster*. *Computation Cluster* wykorzystując *Blockchain Signer* podpisuje wiadomość zawierającą identyfikację obliczeń oraz własny adres Blockchain. Obliczenia opisują identyfikatory trzech wiadomości - *Batch Message*, *Job Message* oraz *Token Message*. Dla potrzeb studium przypadku *Computation Cluster* jest reprezentowany w BalticLSC przez adres Blockchain `0x7bb8e0B0b5f7d3B5 10CAA8CC95196Bbc026A66F7`. Adres ten wynika z przechowywanego przez *Computation Cluster* klucza, który dostarczany jest do modułu *Blockchain Signer* w postaci zmiennej środowiskowej. Moduł *Blockchain signer* zarządza parą kluczy prywatnych i publicznych danego *Computation Cluster*. *Computation Cluster* przy wykorzystaniu modułu *Blockchain signer* podpisuje wiadomości za pomocą klucza prywatnego.

Przykładowo, moduł *Blockchain Signer* jest aktywny na lokalnej maszynie na porcie 3000.

Computation Cluster wykonuje zapytanie *POST* na *localhost:3000/sign-computations* celem otrzymania podpisanej wiadomości z serwisu *Blockchain Signer*.

```
{
  "batchMessageId": "3a7517ac-1f79-4d1f-aaa3-6232b49993fc",
  "jobMessageId": "dfca9a1d-d049-4d7c-88a4-51db9d508610",
  "tokenMessageId": "125e9809-0017-4959-ac05-cb197046faa7"
}
```

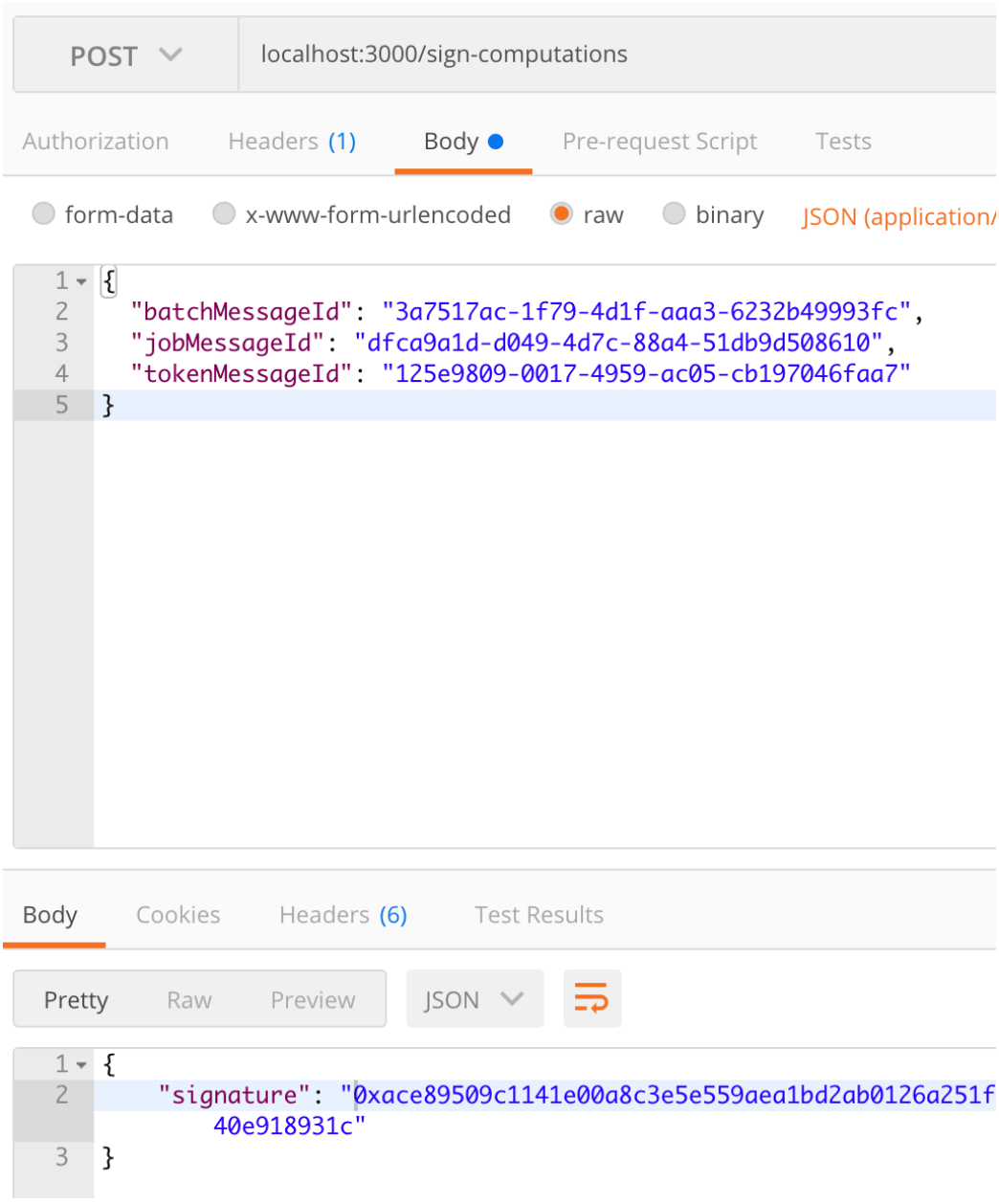
- *BatchMessageId* jest identyfikacją *Batch Message*.
- *JobMessageId* jest identyfikacją *Job Message*.
- *TokenMessageId* jest identyfikacją *Token Message*.

Pierwszy krok testowania za pomocą programu Postman przedstawiono na rysunku 5.1.

Computation Cluster, mając podpisaną wiadomość jako wynik z modułu *Blockchain Signer*, nadaje ją do Blockchain przez *Blockchain Service*. *Blockchain Service* odgrywa dwie role w projekcie:

- może być wykorzystywany przez *Computation Cluster* w celu wysłania wiadomości do Blockchain
- może służyć do weryfikacji obliczeń przez *Batch Manager*.

Blockchain Service dla przykładu jest aktywny na lokalnej maszynie na porcie 3001.



Rysunek 5.1: Pierwszy krok testowania za pomocą Postman - uzyskanie podpisu

Computation Cluster wykonuje zapytanie *POST* na *localhost:3001/submit-computations* celem wysłania podpisanej wiadomości przez serwis *Blockchain Service*.

```
{
  "author": "0x7bb8e0B0b5f7d3B510CAA8CC95196Bbc026A66F7",
  "batchMessageId": "3a7517ac-1f79-4d1f-aaa3-6232b49993fc",
  "jobMessageId": "dfca9a1d-d049-4d7c-88a4-51db9d508610",
  "tokenMessageId": "125e9809-0017-4959-ac05-cb197046faa7",
  "signedTransaction": "0xace89509c1141e00a8c3e5e559ae
  ↪  a1bd2ab0126a251f79250f00cfdcae30a27d50a
  ↪  dbcc0b0bef9a7ec1f852bfe2f56159876237ef5dbd419be9ebdca40e918931c"
}
```

- *SignedTransaction* stanowi podpisaną wiadomość dostarczoną przez *Computation Cluster* jako wynik z modułu *Blockchain Signer*.
- *Author* jest adresem *Computation Cluster*, który wykonał obliczenia.
- *BatchMessageId* stanowi identyfikację *Batch Message*.
- *JobMessageId* stanowi identyfikację *Job Message*.
- *TokenMessageId* stanowi identyfikację *Token Message*.

Odpowiedź o kodzie HTTP 200, oznacza że transakcja została poprawnie włączona do łańcucha Blockchain. Wykonanie obliczeń definiowanych przez wiadomości, których identyfikatory zostały podane powyżej przez *Computation Cluster* o adresie `0x7bb8e0B0b5f7d3B510CAA8CC95196Bbc026A66F7` zostało zapisane na Blockchain.

Drugi krok testowania za pomocą Postman przedstawiono na rysunku 5.2.

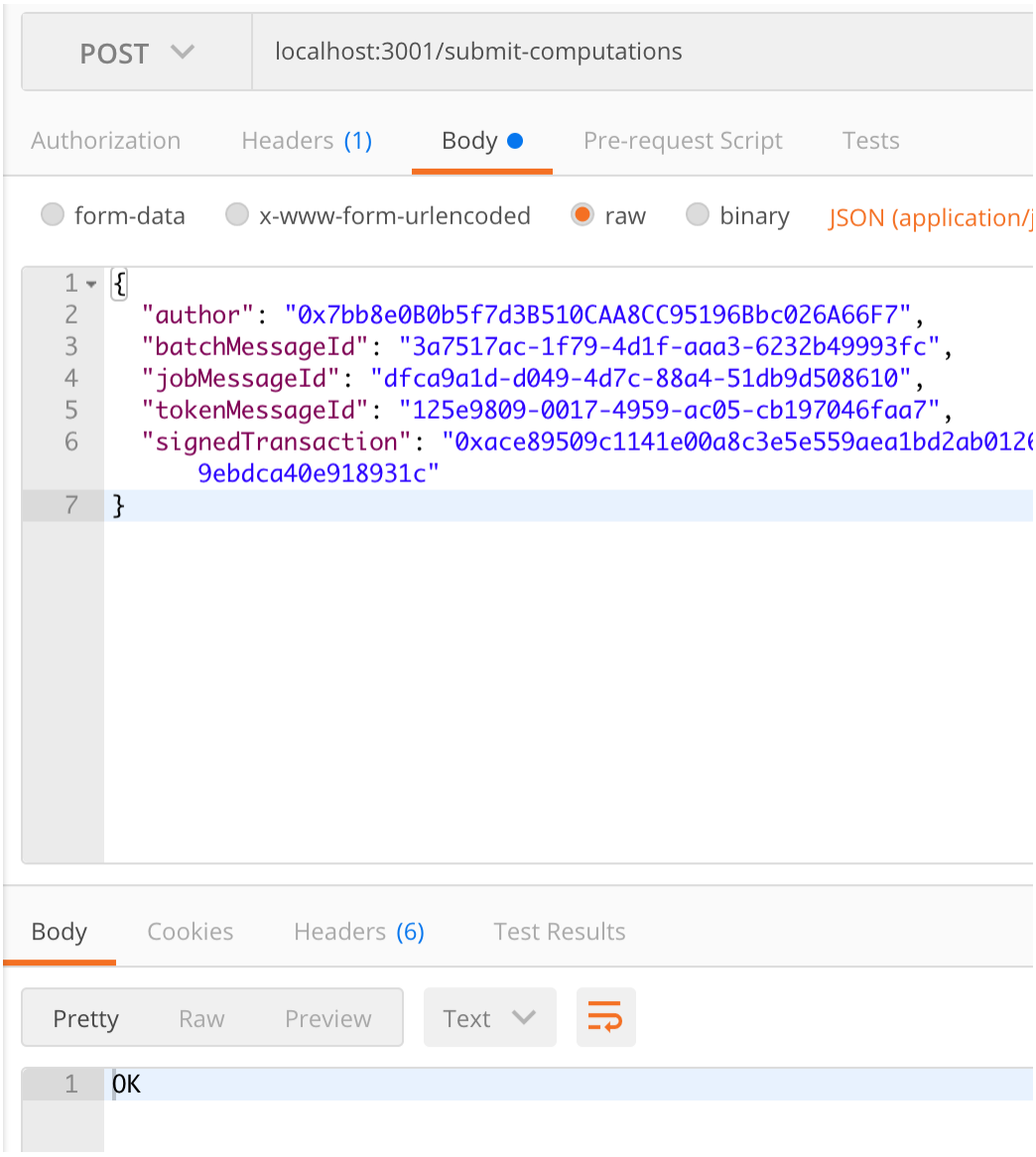
W sieci Blockchain widoczna jest wykonana transakcja:

```
blockchain_node        | 2019-09-30 09:39:16 UTC Imported #10
↪  0x193a...c013 (0 txs, 0.00 Mgas, 0 ms, 0.57 KiB)
blockchain_node        | 2019-09-30 09:39:23 UTC Transaction mined
↪  (hash
↪  0x00f1acb79463480b08ec9ce8fe0a39503028a10f5da853bc988a794f5a36f2b6)
blockchain_node        | 2019-09-30 09:39:23 UTC Imported #11
↪  0xf518...2010 (1 txs, 0.06 Mgas, 0 ms, 0.96 KiB)
```

Po zgłoszeniu obliczeń do *Batch Manager* przez *Computation Cluster*, *Batch Manager* rozpoczyna weryfikację. W tym celu *Batch Manager* wywołuje *Blockchain Service* z identyfikacją wiadomości definiujących obliczenia - *Batch Message*, *Job Message* oraz *Token Message*.

Batch Manager wykonuje zapytanie GET na

```
localhost:3001/verify-computations/3a7517.../dfca9a1d.../125e980...
```

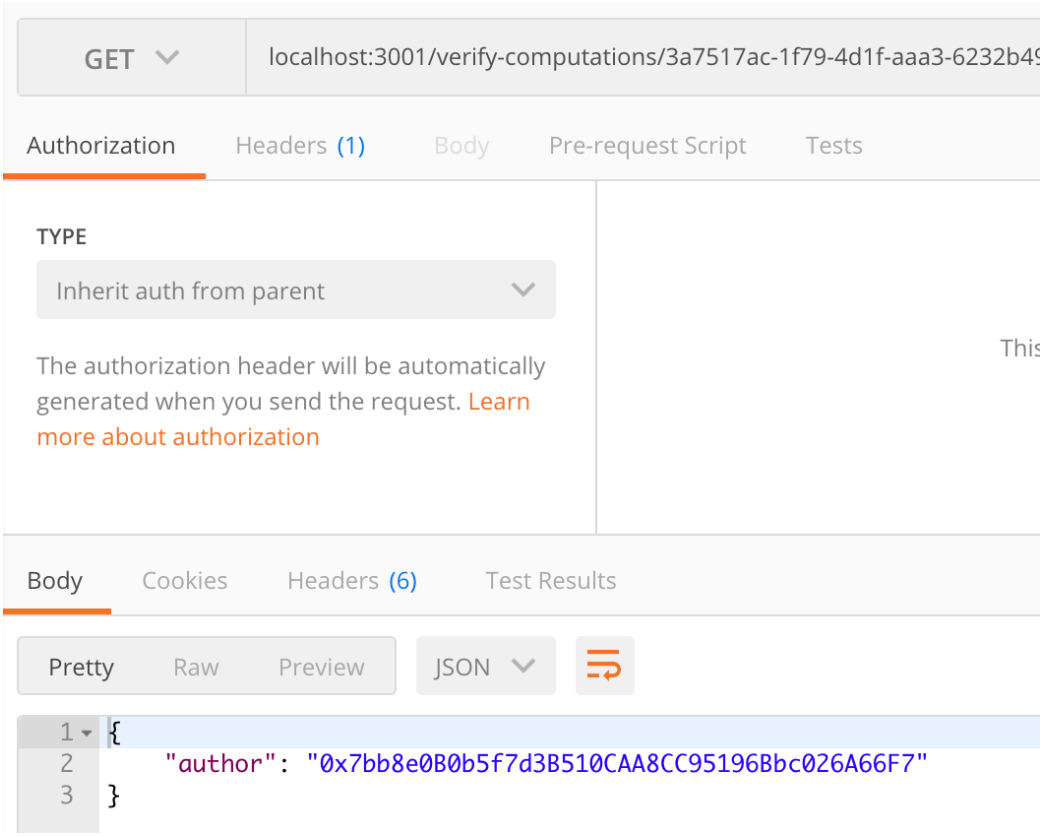


Rysunek 5.2: Drugi krok testowania za pomocą Postman - wysłanie transakcji z sukcesem

przez serwis *Blockchain Service*.

Są to kolejno identyfikatory *Batch Message*, *Job Message* oraz *Token Message*.

Wynikiem wywołania jest adres reprezentujący *Computation Cluster*, który wykonał obliczenia zapisane na Blockchain. W tym przypadku jest to adres: `0x7bb8e0B0b5f7d3B510CAA8CC95196Bbc026A66F7`. Adres *Computation Clu-*



Rysunek 5.3: Trzeci krok testowania za pomocą Postman - zwrócenie adresu autora

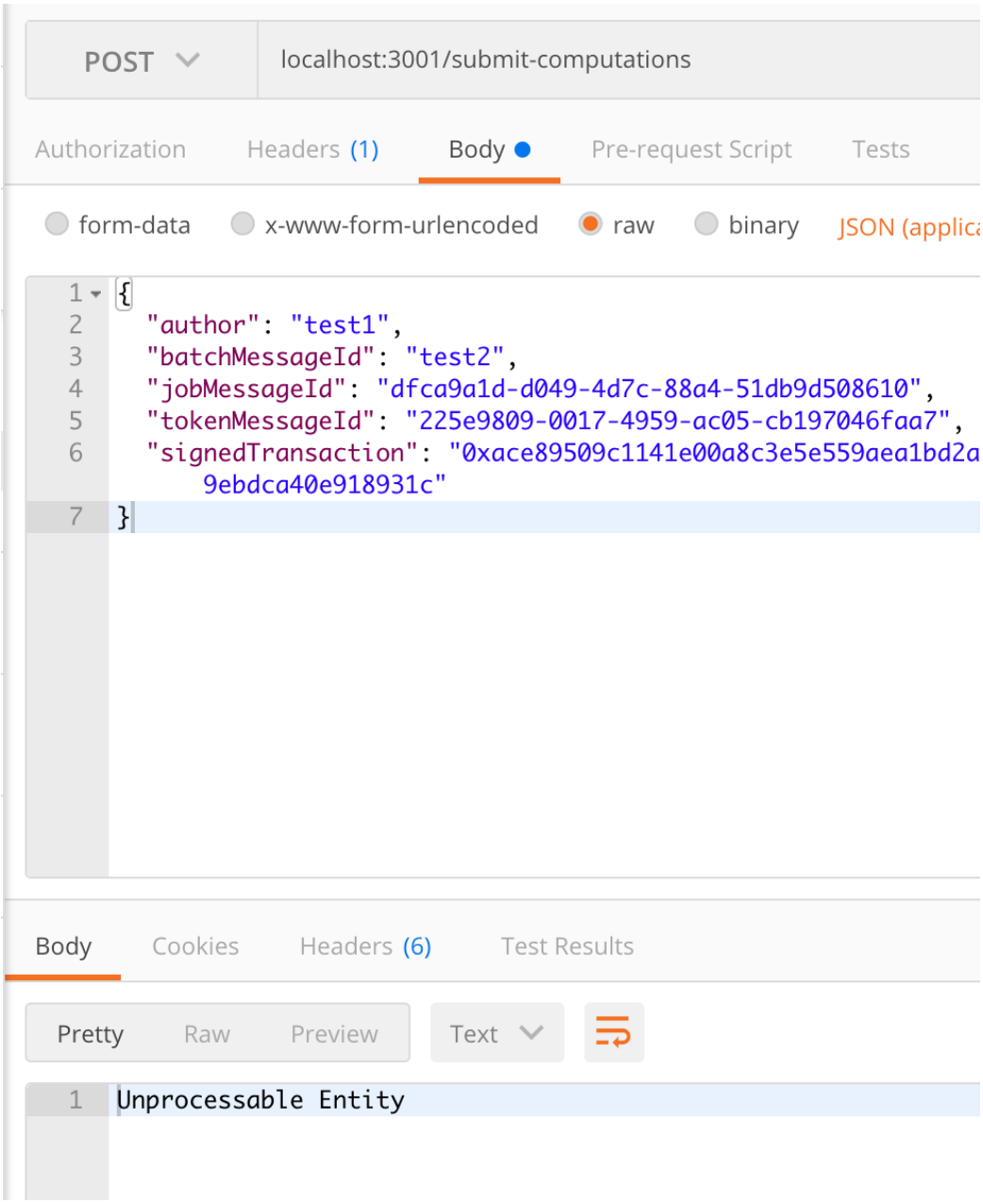
ster został zapisany na sieci Blockchain jako wykonawca obliczeń. Weryfikacja przebiegła pozytywnie.

Trzeci krok testowania za pomocą Postman przedstawiono na rysunku 5.3.

Wywołanie *Blockchain Service* z parametrami, które nie były oryginalnie podpisane przez *Computation Cluster* jest niepoprawne. Wysłanie takiej transakcji skutkuje błędem HTTP 422. Mamy więc gwarancję, że tylko dane, które zostaną oryginalnie podpisane przez *Computation Cluster* będą uwzględnione w łańcuchu Blockchain.

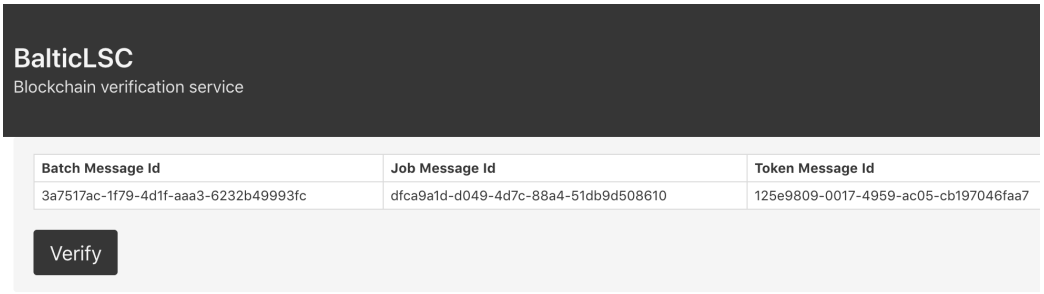
Czwarty krok testowania za pomocą Postman przedstawiony został na rysunku 5.4.

Weryfikacja może być także wykonana poprzez interfejs użytkownika za pomocą modułu *Blockchain viewer*. Fragment interfejsu użytkownika przedstawiono na rysunku 5.5. Użytkownik chciałby sprawdzić czy obliczenia zo-

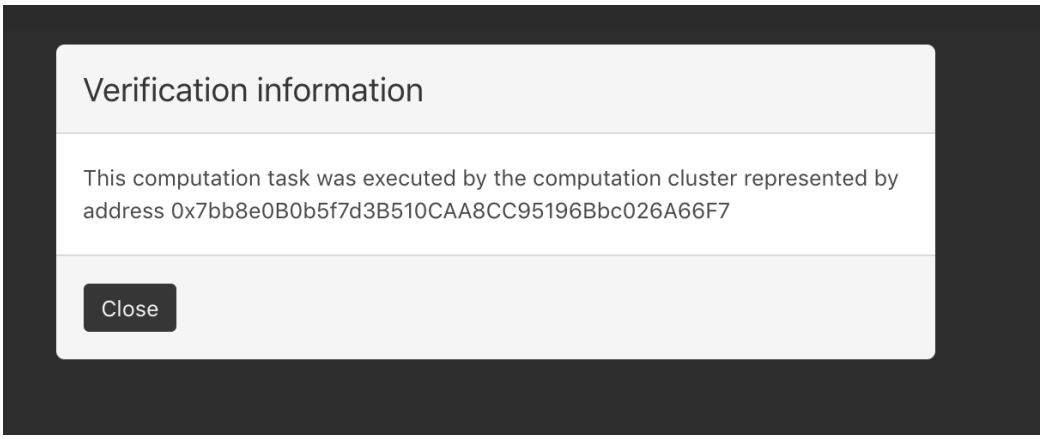


Rysunek 5.4: Czwarty krok testowania za pomocą Postman - wysłanie transakcji bez sukcesu za pomocą Postman

stały wykonane przez dany *Computation Cluster*. BalticLSC dostarcza adres danego *Computation Cluster*, który użytkownik może porównać z tym obec-



Rysunek 5.5: Fragment interfejsu użytkownika przed sprawdzeniem obliczeń



Rysunek 5.6: Fragment interfejsu użytkownika - sprawdzenie *Computation Resource* wykonującego obliczenia *Computation Task Step*

nym na Blockchain.

Dla przykładu moduł *Blockchain viewer* znajduje się na lokalnej maszynie na porcie 3002.

Działa on podobnie jak weryfikacja, którą wykonuje BalticLSC w sposób automatyczny. Po wprowadzaniu identyfikacji wiadomości definiujących obliczenia (w tym przypadku zastosowano wartości stałe) zwraca on adres *Computation Cluster* wykonującego obliczenia.

Fragment interfejsu użytkownika, gdzie weryfikacja przebiegła pozytywnie jest dostępny na rysunku 5.6.

Rozdział 6

Podsumowanie i wnioski

Próba stworzenia modułów korzystających z Blockchain - *Blockchain*, *Blockchain service*, *Blockchain signer* oraz *Blockchain viewer* została zakończona sukcesem. Wykonanie projektu w ramach części praktycznej pracy magisterskiej pozwoliło zatem potwierdzić, że technologia Blockchain może zostać z powodzeniem wykorzystywana do weryfikacji obliczeń rozproszonych. Dowodem na to jest ujęte w rozdziale 5 studium przypadku przedstawiające użycie stworzonych modułów w kontekście systemu BalticLSC.

Zastosowanie systemu Blockchain do weryfikacji obliczeń rozproszonych wydaje się być dobrym rozwiązaniem ze względu na wykorzystaną w nim kryptografię, która gwarantuje bezpieczeństwo wprowadzanych do systemu informacji. Ochrona danych wynika z faktu, że każdy *Computation Cluster* wykonujący obliczenia ma swoją parę kluczy: prywatny i publiczny, które służą do jego identyfikacji lub wysłania nowych transakcji na Blockchain. Rozproszenie sieci Blochchain jako jej cecha umożliwia pracę rozmieszczonych na całym świecie podmiotów korzystających z tej sieci. Jest to szczególnie ważne w przypadku systemu BalticLSC, którego partnerzy, mimo że ulokowani są na terenie różnych krajów Unii Europejskiej, mogą mieć do dostęp do sieci. Poza tym ważną rolę odgrywa fakt, że dane są replikowane na wszystkich węzłach Blockchain, co daje pewność, że nawet w sytuacji, w której odłączy się jedna z maszyn, będą one nadal dostępne dla innych podmiotów. Ponadto Blockchain jest technologią niezależną od systemu BalticLSC, co pozwala na gromadzenie w nim danych, do których nadal będzie możliwy dostęp nawet w przypadku awarii BalticLSC. Pozwala to również na niezależną weryfikację danych przez audyt.

Zaimplementowany projekt magisterski jest w pełni działającym rozwiązaniem, które mogłoby znaleźć zastosowanie w tworzonej sieci BalticLSC. W przyszłości sieć Blockchain mogłaby być rozwijana poprzez dodanie kolejnych *Validator Node* reprezentujących organizacje partnerów biorących udział w

BatlicLSC.

Warto wspomnieć jednak o kilku wadach rozwiązań zastosowanych podczas implementacji projektu. Przede wszystkim Etereum jest dość złożone. Próg wejścia w tę technologię oraz pojęcia w niej wykorzystywane mogą wydać się skomplikowane dla programistów, którzy nigdy nie mieli do czynienia z Blockchain. Obecnie znajduje się na jej temat niewiele publikacji (zwłaszcza w języku polskim), co dodatkowo utrudnia jej wykorzystanie. Innym aspektem jest to, że w technologii Ethereum stosuje się *gas* odpowiadający za opłatę za transakcję. Mimo że stanowi on część architektury Ethereum, nie posiada on w projekcie praktycznego zatosowania. Poza tym *gas* wydaje się stanowić przeszkodę w zrozumieniu projektu oraz ograniczać wydajność tworzenia oprogramowania. Zastosowanie innych technologii w miejsce Ethereum mogłoby rozwiązać ten problem. W wielu aspektach technologia Etereum wymaga dopracowania np. biblioteka web3.js używana przez backend do interakcji z Blockchainem nie zawsze jest godna zaufania.

Warto pamiętać, że technologia Blockchain nie jest nastawiona na częste zmiany w logice biznesowej. Jeżeli zmieni się logika w *Smart Contract CalculationVerification.sol* zainicjalizowana już na Blockchain, należałoby zainicjalizować ją ponownie. Może to stanowić pewną niedogodność, ponieważ wcześniejsze dane będą przechowywana na starej wersji *Smart Contract CalculationVerification.sol*.

Technologia Blockchain jest obecnie stale rozwijana. Możliwe jest zatem, że w przyszłości powstaną nowe rozwiązania lepiej dopasowane do weryfikacji obliczeń rozproszonych.

Tworzenie projektu magisterskiego postawiło przed autorem wyzwanie dokładnego zapoznania się z technologią Blockchain, co było niezbędne do nabycia praktycznych umiejętności tworzenia prywatnych sieci, wykorzystywania języka Solidity do napisania *Smart Contract* oraz inicjalizowania go na Blockchain. Praca ta zmotywowała autora również do poszerzenia wiedzy na temat Ethereum, zwłaszcza w zakresie wykorzystania tej technologii do tworzenia prywatnych łańcuchów Blockchain. Konstrukcja stworzonego *Smart Contract CalculationVerification.sol* wymagała zrozumienia istoty funkcjonalności *ecrecover* umożliwiającej odtworzenie adresu podmiotu podpisującego wiadomość wewnątrz *Smart Contract*. Wyzwaniem dla autora było również stworzenie i konteneryzowanie aplikacji w języku Typescript oraz zbudowanie integracji z siecią Blockchain z poziomu serwera i interfejsu użytkownika. Praca nad BalticLSC pozwoliła też zapoznać się z definicją i znaczeniem systemów obliczeń rozproszonych oraz z zasadami prawidłowego prowadzenia projektu, w tym definiowania wymagań, tworzenia diagramów. Rozwiązania wdrożone do architektury BalticLSC umożliwiły również przeanalizowanie szczegółów technologii mikroserwisów oraz sposobów zastosowania ich

w większych projektach. Nabyte umiejętności i wiedza mogą zaowocować w przyszyłości możliwością tworzenia nowoczesnych aplikacji internetowych łączących się z Blockchain.

Dodatek A

Budowanie i uruchamianie obrazów Docker

A.0.1 Moduł Blockchain

Komenda do zbudowania obrazu Docker z węzłem Blockchain jest następująca:

```
docker build --no-cache -t baltic-lsc-node .
```

Komenda do uruchomienia obrazu Docker jest następująca:

```
docker run -p 8545:8545 -p 8546:8546 baltic-lsc-node
```

Po uruchomieniu obrazu węzeł Blockchain zostanie włączony na maszynie lokalnej.

Interakcja z węzłem Blockchain odbywa się przez protokół JSON-RPC [28]. Węzeł nasłuchuje na połączenia na porcie 8545.

Inicjalizacja Smart Contractu odbywa się poprzez obraz Dockerowy, wykorzystując bibliotekę Truffle. Konieczne jest, aby węzeł Blockchain nasłuchiwał na połączenia protokołu JSON-RPC.

Komenda do zbudowania obrazu Docker jest następująca:

```
  docker build --no-cache -t baltic-blockchain-deployer .
```

Komenda do uruchomienia obrazu Docker jest następująca:

```
docker run -e "HOST= http://docker.for.mac.host.internal:8545" -e
↪  "PRIVATE_KEY=xxx" baltic-blockchain-deployer
```

Po uruchomieniu powyższej komendy obraz Dockerowy zainicjalizuje Smart Contracty na węźle Blockchain.

Zmienne, środowiskowe których należy użyć:

- *HOST* - definiuje adres, pod którym nasłuchuje węzeł Blockchain.
- *PRIVATE_KEY* - definiuje klucz prywatny konta administracyjnego nadającego transakcję do Blockchain.

A.0.2 Moduł Blockchain signer

Komenda do zbudowania obrazu Docker jest następująca:

```
docker build --no-cache -t baltic-blockchain-signer .
```

Komenda do uruchomienia obrazu Docker jest następująca:

```
docker run --env-file ./.env -p 3000:3000 baltic-blockchain-signer
```

Przed uruchomieniem należy ustawić zmienne środowiskowe (w pliku .env):

- *SIGNER_PRIVATE_KEY* - jest kluczem prywatnym *Computation Cluster*.

Moduł *Blockchain service* będzie nasłuchiwał na porcie 3000.

A.0.3 Moduł Blockchain service

Komenda do zbudowania obrazu Docker jest następująca:

```
docker build --no-cache -t baltic-blockchain-service .
```

Komenda do uruchomienia obrazu Docker jest następująca:

```
docker run --env-file ./.env -p 3001:3001 baltic-blockchain-service
```

Przed uruchomieniem należy ustawić zmienne środowiskowe (w pliku .env):

- *CONTRACT_ADDRESS* - jest adresem zainicjalizowanego Smart Contract.
- *SERVICE_ACCOUNT_PRIVATE_KEY* - jest kluczem prywatnym konta administracyjnego wysyłającego transakcje na Blockchain.
- *BLOCKCHAIN_NODE_URL* - jest adresem węzła Blockchain.

Moduł *Blockchain service* będzie nasłuchiwał na porcie 3001.

A.0.4 Moduł Blockchain viewer

Budowanie obrazu Docker

```
docker build --no-cache --build-arg REACT_APP_CONTRACT_ADDRESS=x
↪  --build-arg PORT=3002 -t baltic-blockchain-viewer .
```

- *REACT_APP_CONTRACT_ADDRESS* - stanowi zainicjalizowanego Smart Contractu
- *PORT* - to port aplikacji

Uruchamianie obrazu Docker

```
docker run -p 3002:3002 baltic-blockchain-viewer
```

Po uruchomieniu interfejs użytkownika będzie dostępny na porcie 3002 na lokalnej maszynie.

Dodatek B

REST API zaimplementowanych modułów

B.0.1 Moduł Blockchain signer

POST /submit-computations

Żądanie zastosowe w celu podpisania wiadomości przez *Computation Cluster*, na które składają się wykonane obliczenia. Wynikiem jest podpisana wiadomość zawierająca adres *Computation Cluster* i identyfikator wiadomości obliczeń.

Żądanie:

```
{
  "batchMessageId": "3a7517ac-1f79-4d1f-aaa3-6232b49993fc",
  "jobMessageId": "dfca9a1d-d049-4d7c-88a4-51db9d508610",
  "tokenMessageId": "125e9809-0017-4959-ac05-cb197046faa7"
}
```

- *batchMessageId* jest identyfikacją *Batch Message*
- *jobMessageId* jest identyfikacją *Job Message*
- *tokenMessageId* jest identyfikacją *Token Message*

Odpowiedź: 200 OK

```
{
  "signature":
  ↪  "0xace89509c1141e00a8c3e5e559aea1bd2ab0126a251f79250f00c
  fdcae30a27d50adbcc0b0bef9a7ec1f852bfe2f56159876237ef5dbd419
  be9ebdca40e918931c"
}
```

- *signature* - podpisana wiadomość Blockchain

B.0.2 Moduł Blockchain service

Dokumentacja API

POST /submit-computations
Żądanie mające zastosowanie w wysyłaniu transakcji na Blockchain:

```
{
  "author": "0x7bb8e0B0b5f7d3B510CAA8CC95196Bbc026A66F7",
  "batchMessageId": "3a7517ac-1f79-4d1f-aaa3-6232b49993fc",
  "jobMessageId": "dfca9a1d-d049-4d7c-88a4-51db9d508610",
  "tokenMessageId": "125e9809-0017-4959-ac05-cb197046faa7",
  "signedTransaction":
  ↪  "0xace89509c1141e00a8c3e5e559aea1bd2ab0126a251f79250f00cfdcae3
  ↪  0a27d50adbcc0b0bef9a7ec1f852bfe2f56159876237ef5dbd419be9ebdca4
  ↪  0e918931c"
}
```

- *signedTransaction* - podpisana wiadomość dostarczona przez *Computation Cluster* jako wynik z modułu *Blockchain Signer*
- *author* - adrese *Computation Cluster*, który wykonał obliczenia
- *batchMessageId* - identyfikacja *Batch Message*
- *jobMessageId* - identyfikacja *Job Message*
- *tokenMessageId* - jest identyfikacja *Token Message*

Odpowiedzi:

- 200 OK - jeżeli transakcja na Blockchainie powiodła się
- 422 Bad Request - jeżeli transakcja na Blockchainie nie powiodła się

GET /verify-computations
Żądanie stosowane w celu weryfikacji danego *Computation Task Step*:

```
/verify-computations/:batchMessageId/:jobMessageId/:tokenMessageId
```

- *batchMessageId* - identyfikacja *Batch Message*
- *jobMessageId* - identyfikacja *Job Message*
- *tokenMessageId* - identyfikacja *Token Message*

Odpowiedź:

```
{
  "author": "0x7bb8e0B0b5f7d3B510CAA8CC95196Bbc026A66F7"
}
```

- *author* - adres identyfikujący *Computation Cluster*

Bibliografia

[1] Michał Śmiałek, Zrozumieć UML 2.0, Helion, 2005.

[2] Chris Dannen, Introducing Ethereum and Solidity, Aprees, 2017.

[3] Deloitte's 2019 Global Blockchain Survey, 2019.

[4] Anders Brownworth, Github, https://andersbrownworth.com/blockchain/blockchain

[5] Wikipedia, Blockchain, https://pl.wikipedia.org/wiki/Blockchain

[6] Wikipedia, Etherum White Paper, https://github.com/ethereum/wiki/wiki/White-Paper

[7] Google Trends https://trends.google.com/

[8] Mikael Häggström, WikiCommons, https://commons.wikimedia.org/wiki/File:Market_capitalizations_of_cryptocurrencies.svg

[9] The next web, These are top 10 programming languages in blockchain , https://thenextweb.com/hardfork/2019/05/24/javascript-programming-java-cryptocurrency/

[10] EtherScaner, https://etherscan.io/token/0x2a98f128092abbadef25d17910ebe15b8495d0c1

[11] BitcoinWiki, ERC20, https://en.bitcoinwiki.org/wiki/ERC20

[12] Proof of Stake FAQ, Github, https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ

[13] Github, https://github.com/beched/abi-decompiler

[14] Private Transactions, Github, https://wiki.parity.io/Private-Transactions

[15] Pluggable-Consensus Aura, Github, https://wiki.parity.io/Pluggable-Consensus.html#aura

[16] Validator Set Code, Github, https://github.com/parity-contracts/kovan-validator-set

[17] What is Golem?, https://docs.golem.network/#/About/What-is-Golem

[18] DrawIo, https://www.draw.io/

[19] Node.js, https://nodejs.org/en/

[20] TypeScript, https://www.typescriptlang.org/

[21] State of Js, https://2019.stateofjs.com/

[22] Geth, https://geth.ethereum.org/

[23] Truffle, https://www.trufflesuite.com/

[24] Dev.to, Which programming language is best for blockchain?, https://dev.to/duomly/which-programming-language-is-the-best-for-blockchain-all

[25] Docker, https://www.docker.com/

[26] Chain Specification, https://wiki.parity.io/Chain-specification

[27] Cross-Origin Resource Sharing, https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

[28] JSON RPC API, https://wiki.parity.io/JSONRPC

[29] Docker compose, https://docs.docker.com/compose/

[30] Baltic Large Scale Computing, https://projects.interreg-baltic.eu/projects/balticlsc-172.html

[31] Baltic LSC, https://www.balticlsc.eu/

[32] Baltic LSC Enviorment vision, https://www.balticlsc.eu/wp-content/uploads/2019/07/03.1_BalticLSC_Environment_Vision.pdf

[33] BalticLSC Platform Architectural Vision, https://www.balticlsc.eu/wp-content/uploads/2019/07/04.1_BalticLSC_Platform_Architectural_Vision-3.pdf

[34] Kubernetes, https://kubernetes.io/

[35] Rancher, https://rancher.com/

[36] BalticLSC Software Architectural Vision, https://www.balticlsc.eu/wp-content/uploads/2019/07/05.1_BalticLSC_Software_Architectural_Vision-3.pdf

[37] Diagramy BalticLSC, https://www.balticlsc.eu/model/