

Politechnika Świętokrzyska		
Wydział elektrotechniki, automatyki i informatyki		
Projekt Systemy Odporne na Błędy		
Studia: Stacjonarne II stopnia	Kierunek: Informatyka	Autorzy: Wojciech Lisowski Tomasz Szymański Michał Zwierzyński Grupa: 2ID22B
Temat: BitCoin: Rejestr transakcji.		

Historia zmian

1.0 15-06-2020		
1.1 16-06-2020	Michał Zwierzyński	Dodano 5 scenariusz i zaktualizowano wnioski

1. Wstęp

Tematem projektu jest system oparty na sieci blockchain realizujący odporność na błędy. Aplikacja dzieli się na aplikację klienta wykorzystującą bibliotekę React.js oraz aplikację serwera opartą na Node.js. Poszczególne aplikacje będąc peerami łączy się z innymi peerami. Do komunikacji wykorzystano bibliotekę WebSocket. W rozdziale 2 zawarto opis sieci blockchain. Rozdział 3 zawiera opis implementacji projektu. W rozdziale 4 przetestowano aplikację pod kątem odporności na błędy. Rozdział 5 zawiera wnioski z przeprowadzonych testów.

2. Działanie rozproszonej sieci Blockchain

W sieci blockchain dane przechowywane są w formie łańcucha bloków (blockchain) połączonych ze sobą za pomocą zaszyfrowanej wiadomości. Dane te mogą dotyczyć na przykład transakcji dokonywanych za pomocą kryptowalut. Zdecentralizowana sieć w jakiej funkcjonuje blockchain nie posiada żadnego centralnego zarządcy, który akceptuje lub odrzuca transakcje dostające się do sieci. Algorytm konsensusu (consensus protocol) zapewnia, że poszczególne urządzenia znajdujące się w sieci muszą uzgadniać między sobą ważność danej transakcji. Algorytm ten jest jednocześnie jedną z najważniejszych technologii wchodzących w skład blockchain. Proof of Work jest algorytmem konsensusu wprowadzonym na potrzeby Bitcoina. Proces ten jest szerzej znany jako górnictwo (mining). Użytkownicy sieci udostępniają czas i moc obliczeniową, które służą do rozwiązania zadania matematycznego zaproponowanego przez sieć. Pierwszy kto rozwiąże problem posiada zdolność do weryfikacji przeprowadzonej transakcji i generuje nowy blok, otrzymując w zamian nagrodę.

3. Implementacja aplikacji

Aplikacja dzieli się na aplikację kliencką zaimplementowaną w środowisku JavaScript React.js. Aplikacja dostarcza interfejsu użytkownika umożliwiając zarządzanie uproszczoną implementacją blockchained. Klient w celu pozyskania danych łączy się serwerem. Obie aplikacje tworzą peer. Ilość uruchomionych peerów jest dowolna i mogą one się łączyć z pozostałymi i wymieniać informacje. Implementacja blockchained została przedstawiona jako

system wykonywania transakcji kryptowalutą. Aplikacja dostarcza następujących funkcji: dodawanie transakcji (ze wskazaniem adresu i kwoty), wydobywanie bloku z transakcjami w toku, przeglądanie listy bloków oraz transakcji.

3.1 Klient

Aplikacja kliencka została zaimplementowana z użyciem biblioteki React.js. Biblioteka React.js dostarcza narzędzi do tworzenia jednostronicowych aplikacji internetowych w formie komponentów.

W aplikacji klienckiej znajdują się 4 główne komponenty: *dashboard*, *transaction*, *blockchain*, *pool*. Każdy główny komponent zawiera funkcjonalny widok. W dashboardzie zawarto proste podsumowanie wykonanych transakcji wraz z identyfikatorem portfela i stanem konta. Widok *transaction* zawiera dwa pola tekstowe: dla identyfikatora portfela oraz kwoty. Poniżej pól znajduje się przycisk dodający nową transakcję – użytkownik informowany jest stosownym komunikatem. W widoku *blockchain* znajduje się lista bloków identyfikowana kodem hashującym z możliwością przeglądania transakcji, które są dołączone do danego bloku. Ostatni z widoków, *pool* przedstawia listę wykonanych transakcji przez peer'ów. Poniżej można dokonać miningu, który zapisze transakcje – przy czym musi być co najmniej jedna w kolejce – w formie bloku dołączając go do łańcucha. Widoki zostały zdefiniowane w pliku *App.tsx* z użyciem komponentów *Switch* i *Route* pochodzących z biblioteki *react-router-dom*. Biblioteka dostarcza stronicowania i umożliwia przełączanie się między widokami dzięki adresom zapisanym w ścieżce URL. Plik *App.tsx* zawiera prosty komponent funkcyjny, bez stanowy, oznacza to, że nie przechowuje stanów, które są zmieniane w trakcie działania aplikacji – nie zmienia się w trakcie działania programu.

```
function App() {  
  return (  
    <Switch>  
      <Route path="/dashboard" exact component={Dashboard} />  
      <Route path="/transaction" exact component={Transactions} />  
      <Route path="/blockchain" exact component={Blockchain} />  
      <Route path="/pool" exact component={Pool} />  
      <Redirect to="/dashboard" />  
    </Switch>  
  );  
}
```

Listing 1 Komponent funkcyjny App

Widok dashboard jest lokalizowany przez adres /dashboard. Widok ten został przedstawiony w komponencie funkcyjnym *Dashboard.tsx*.

```
const Dashboard: React.FC<RouteComponentProps<any>> = ({ location }) => {
  const classes = useStyles();
  const [dense, setDense] = React.useState(false);
  const [getPk, setPk] = React.useState("");
  const [getBalance, setBalance] = React.useState("");
  const [getHistory, setHistory] = React.useState<IHistory[]>([]);
  React.useEffect(() => {
    const fetchData = async () => {
      console.log(`${process.env.REACT_APP_HTTP_PORT}`);
      const r1 = await axios(`http://localhost:${process.env.REACT_APP_HTTP_PORT}/public-key`);
      setPk(r1.data.publicKey);
      const r2 = await axios(`http://localhost:${process.env.REACT_APP_HTTP_PORT}/balance`);
      setBalance(r2.data.balance);
      const r3 = await axios(`http://localhost:${process.env.REACT_APP_HTTP_PORT}/history`);
      setHistory(r3.data.history);
    };
    fetchData();
  }, [setPk]);
  return (
    <React.Fragment>
    ...
  )
}
```

Listing 2 Fragment komponentu funkcyjnego Dashboard

Komponent posiada zapisane 4 stany przy pomocy React hooka `useState`. Stany służą do zapisania danych pobieranych z API za pomocą biblioteki *axios*. Dane te dotyczą klucza publicznego – primary key, stanu konta – balance i history – historii transakcji. Adres API wykorzystuje port zapisany w zmiennej środowiskowej tak, by można było zidentyfikować serwer w przypadku równoległego działania wielu aplikacji. W renderowanej części komponentu użyto zagnieżdżonego komponent *main.tsx* zawierający nagłówek wraz z menu zawierającym listę odnośników do danych widoków.

```

return (
  <div className={classes.root}>
    <AppBar position="static">
      <Toolbar>
        <IconButton edge="start" className={classes.menuBut-
ton} onClick={handleClick} color="inherit" aria-label="menu">
          <MenuIcon />
        </IconButton>
        <Typography variant="h6" className={classes.title}>
          Blockchain App : {name}
        </Typography>
      </Toolbar>
    </AppBar>
    <Menu
      id="menu"
      open={Boolean(anchorEl)}
      anchorEl={anchorEl}
      keepMounted
      onClose={handleClose}
    >
      <MenuItem><Link className={classes.link} to="/">Dash-
board</Link></MenuItem>
      <MenuItem><Link className={classes.link} to="/transac-
tion">New transaction</Link></MenuItem>
      <MenuItem><Link className={classes.link} to="/block-
chain">Blockchain</Link></MenuItem>
      <MenuItem><Link className={classes.link} to="/pool">Transac-
tion pool</Link></MenuItem>
    </Menu>
  </div >

```

Listing 3 Fragment metody renderującej komponentu Main

Elementy jak `Menu` oraz `MenuItem` dostarcza gotowa biblioteka *Material-UI* zawierająca gotowe komponenty przydatne przy tworzeniu interfejsów użytkownika. Komponent *Link* jest powiązany z konkretnym widokiem za pomocą właściwości o nazwie *to*. W komponencie *transaction* występuje metoda `POST` do tworzenia nowej transakcji, która jest przesyłana do API. Metoda ta przyjmuje dwa parametry: klucz publiczny – adres odbiorcy oraz kwota – suma jaka zostanie zapisana użytkownikowi o wskazanym kluczu. Podobnie jak w komponencie *Main* wykorzystywane są gotowe komponenty *Material-UI* są to przede wszystkim *TextField*, który jest polem tekstowym. W React występuje tak zwane dwustronne

wiązanie. To co zostało wpisane w pole tekstowe jest zapisane w stanie, wartość value pobierana jest właśnie z tego stanu.

```
return (
  <React.Fragment>
    <Main name={location.pathname.replace('/', '')} />
    <Snackbar open={open} autoHideDuration={6000} onClose={handle-
Close}>
      <Alert onClose={handleClose} severity="success">
        Tx has been send
      </Alert>
    </Snackbar>
    <div className={classes.root}>
      <Container maxWidth="md">
        <Grid container spacing={3}>
          <Grid item xs={12}>
            <TextField
              className={classes.text}
              id="standard-read-only-input"
              label="Destination address"
              onChange={handleAddress}
              value={address}
            />
          </Grid>
          <Grid item xs={12}>
            <TextField
              className={classes.text}
              id="standard-read-only-input"
              label="Amount"
              type="number"
              onChange={handleAmount}
              value={amount}
            />
          </Grid>
          <Grid item xs={12}>
            <Button variant="contained" color="pri-
mary" onClick={send} disabled={amount.length === 0 || ad-
dress.length === 0}>
              Send
            </Button>
          </Grid>
        </Grid>
      </Container>
    </div>
  </React.Fragment>
)
```

Listing 4 Fragment metody renderującej komponentu Transactio

Niektóre komponenty jak na przykład komponent *Blockchain* wykorzystują listy do renderowania powtarzających się elementów. Ten schemat ma zastosowanie w przypadku wyświetlania listy bloków pobieranych z API. Lista ta której typ zdefiniowany jest przy użyciu interfejsów *IChain* oraz *ITransaction* zapisana jest w stanie `getBlockchain`.

```
{getBlockchain?.map((e, i) => (  
    <React.Fragment key={i}>  
        <Grid item xs={4}>  
            <Card className={classes.root} variant="outlined">  
                <CardContent className={classes.root}>  
                    <Typography  
                        variant="body2"  
                        color="textSecondary"  
                        component="p"  
                    >  
                        <strong>HashCode: </strong>  
                        {e?.hash}  
                    </Typography>  
                    <Typography  
                        variant="body2"  
                        color="textSecondary"  
                        component="p"  
                    >  
                        <strong>Timestamp: </strong>  
                        {e?.timestamp}  
                    </Typography>  
                </CardContent>  
                <CardActions>  
                    <Button variant="contained" color="secondary">  
                        Show Transactions  
                    </Button>  
                </CardActions>  
            </Card>  
        </Grid>  
    </React.Fragment>  
)) }
```

Listing 5 Zawartość funkcji renderującej w komponencie Blockchain. Fragment zawierający renderowanie listy

```

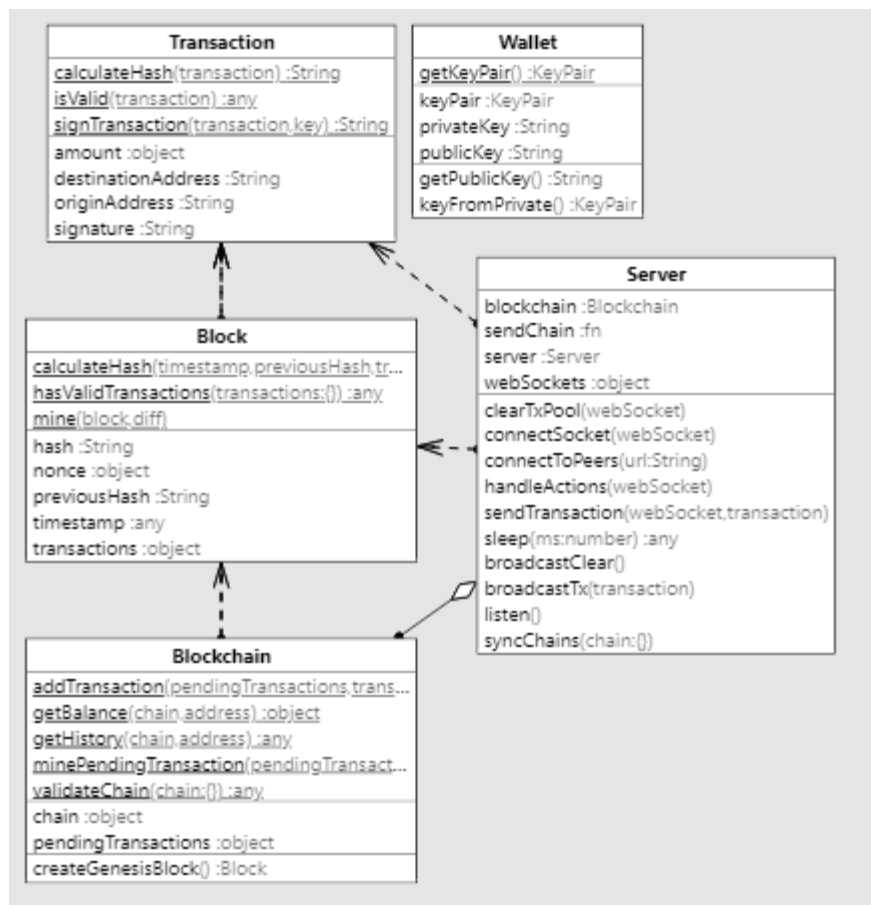
interface ITransaction {
  signature: string;
  amount: number;
  originAddress: string;
  destinationAddress: string;
}
interface IChain {
  nonce: number;
  timestamp: Date;
  transactions: ITransaction[];
  hash: string;
}

```

Listing 6 Intefejsy definiujące typ łańucha i transakcji

3.2 Serwer

Aplikacja serwera działająca w środowisku *Node.js* jest bardziej złożona względem aplikacji klienta, której głównym celem jest dostarczanie interfejsu użytkownika. Protokołem łączącym aplikacje serwera i klienta jest protokół REST. Część endpointów, oraz to jakie pełnią funkcje zostały opisane częściowo w podrozdziale 3.1. Do działania serwera REST’owego wykorzystywana jest biblioteka *Express.js* Każdy serwer posiada ponadto zdolność komunikacji z innymi serwerami. Pełni więc rolę tak zwanego peera w rozproszonej sieci równorzędnych aplikacji P2P. Do zbudowania architektury wykorzystano szybki protokół WebSocket, który dostarczony jest w formie biblioteki środowiska Node.js *ws*. W klasie *Server* znajduje się implementacja sieci P2P. Pozostałe pliki zawierają klasy związane z implementacją i działaniem sieci Blockchain. Główną klasą jest klasa *Blockchain* agregująca klasy pomocnicze przy działaniu sieci blockchain: *Transaction* -zawierający metody służące do obsługi transakcji, *Block* – zawierający metody umożliwiające zarządzanie pojedynczym blokiem. Osobną klasą jest klasa *Wallet*, która pełni rolę tworzenia kluczy i prywatnych i publicznych, za pomocą których można identyfikować transakcję oraz ją podpisywać. Na rysunku 1 zawarto diagram klas aplikacji serwera.



Rysunek 1 Diagram klas aplikacji serwera

Po uruchomieniu serwera w pliku *index.ts* tworzony jest obiekt klasy Express tworzący serwer REST, a następnie tworzone są obiekty klas Server, Blockchain oraz Wallet. Do serwera rest za pośrednictwem zmiennej *app* definiowane są ustawienia CORS oraz poszczególne endpointy, które przedstawia listing 7.

```

const POST_TRANSACTIONS = "/transact";
const POST_MINE = "/mine-transactions";
const GET_BALANCE = "/balance";
const GET_CHAIN = "/chain";
const GET_PUBLIC_KEY = "/public-key";
const GET_TRANSACTIONS = "/transactions";
const GET_CHAIN_VALIDATION = "/validate";
const GET_HISTORY = "/history";
  
```

Listing 7 Lista stałych zawierających adresy endpointów

POST_TRANSACTIONS – metoda przyjmuje dwa parametry *destinationAddress* oraz *amount*. Tworzy obiekt klasy *Transaction* w konstruktorze, którego umieszczane są oba parametry wraz z kluczem publicznym należącego do peera. Transakcja zostaje podpisana

metodą *signTransaction*. Do podpisania wymagany jest klucz przechowywany na serwerze. Popisana transakcja zostaje dodana do listy transakcji oczekujących (transaction pool) poprzez metodę klasy Blockchain – *addTransaction*. Serwer P2P dokonuje rozgłoszenia nowo dodanej transakcji poprzez metodę *broadcastTx* do pozostałych peerów. W odpowiedzi do zapytania endpoint zwraca klientowi listę transakcji.

POST_MINE – metoda wykonuje mining transakcji zapisanych w liście transakcji oczekujących. Polega to na wykonaniu stosownego algorytmu zaproponowanego przez sieć. Po wykonaniu miningu nowy blok jest dołączany do łańcucha bloków. Łańcuch jest rozgłaszany do pozostałych peerów, a lista transakcji oczekujących czyszczona – polecenie to również jest rozgłaszane do pozostałych peerów. Sprawia to, że lista transakcji oczekujących jest wspólna dla peerów. W odpowiedzi do zapytania endpoint zwraca klientowi listę transakcji.

GET_BALANCE – metoda wykonuje metodę klasy Blockchain *getBalance*, która zwraca saldo dla danego klucza publicznego.

GET_CHAIN - metoda zwraca łańcuch bloków z listą przypisanych transakcji.

GET_PUBLIC_KEY – metoda zwraca klucz publiczny należący do peera

GET_TRANSACTION – metoda zwraca listę transakcji oczekujących

GET_CHAIN_VALIDATION – metoda sprawdza poprawność blockchaina.

GET_HISTORY – metoda zwraca listę dokonanych transakcji w których uczestniczył peer o zapisanym na serwerze kluczu publicznym.

W klasie Server w konstruktorze tworzony jest obiekt klasy WebSocket, do którego przekazywany jest port zdefiniowany w zmiennej środowiskowej. Metoda *listen* nasłuchuje połączeń z serwerem. W przypadku nawiązania połączenia obsługa przekazywana jest do metody *connectSocket* przedstawionej na listingu 8.

```
private connectSocket(webSocket: WebSocket): void {
    this.webSockets.push(webSocket);
    console.log('Peer connected: ' + webSocket.url);
    this.handleActions(webSocket);
    this.sendChain(webSocket, this.blockchain.chain);
}
```

Listing 8 Metoda obsługująca połączenie

Peer po uruchomieniu serwera próbuje nawiązać połączenie z pozostałymi peerami zdefiniowanymi w zmiennej środowiskowej PEERS. Działanie metody obsługującej połączenie polega na wpisaniu obiektu `WebSocket` z adresem url do peera do tablicy `webSockets`. Ta tablica jest następnie rozgłaszana na przykład do metod rozgłaszających i innych operacji. Obiekt ponadto przekazywany jest do metody `handleActions` przedstawionej na listingu 9, gdzie następuje obsługa przesyłanych sygnałów. Sygnały są zdefiniowane w stałych obiektu `actions`, którego przedstawiono na listingu 10.

```
private handleActions(webSocket: WebSocket): void {
    webSocket.on('message', (message: string) => {
        const data = eval('(' + message.toString() + ')');
        switch (data.type) {
            case actions.ADD_TRANSACTION:
                Blockchain.addTransaction(this.blockchain.pendingTransactions, data.data);
                break;
            case actions.SEND_CHAIN:
                try {
                    if (Blockchain.validateChain(data.data)) {
                        if (data.data.length <= this.blockchain.chain.length) {
                            throw new Error("Chain is not longer than current chain. Not replacing.")
                        } else {
                            this.blockchain.chain = data.data;
                        }
                    } else {
                        throw new Error("Chain is not valid");
                    }
                } catch (error) {
                    return error;
                } break;
            case actions.CLEAR_TX_POOL:
                this.blockchain.pendingTransactions = [];
                break;
        }
    })
}
```

Listing 9 Metoda obsługująca akcje

```
const actions = {
    ADD_TRANSACTION: 'ADD_TRANSACTION',
    SEND_CHAIN: 'SEND_CHAIN',
    CLEAR_TX_POOL: 'CLEAR_TX_POOL'
}
```

Listing 10 Metoda zawierająca listę akcji

ADD_TRANSACTION - metoda dodaje transakcje do listy transakcji oczekujących zdefiniowanych w liście obiektu klasy Blockchain

SEND_CHAIN – metoda w pierwszej kolejności poddaje walidacji odebrany łańcuch. Walidacja polega na sprawdzeniu, czy każdy blok następujący po sobie zawiera w hashCode zaszyfrowaną wiadomość identyfikującą poprzedni blok. Jeśli łańcuch jest prawidłowy to sprawdzana jest jego długość. Gdy łańcuch jest takiej samej długości lub dłuższy niż łańcuch lokalny to jest od nadpisywany, w przeciwnym razie odrzucany.

CLEAR_TX_POOL – metoda czyści listę transakcji oczekujących.

W metodzie *handleActions* dana akcja jest najpierw deserializowana do formatu JSON. Zanim wykonana zostanie jedna z akcji to należy ją zainicjować. Służą do tego metody *sendChain*, *sendTransaction* oraz *clearTxPool*. Każda metoda przesyła zserializowany obiekt zawierający typ wiadomości *type* oraz samą wiadomość *data*. Pozostałe w klasie metody to *broadcastClear* i *broadcastTx*, które dla wszystkich zdefiniowanych w tablicy peerów wykonuje akcje czyszczenia listy transakcji oczekujących oraz rozsyłające nową transakcję.

Klasa *Blockchain* skupia w sobie metody związane stricte z zarządzaniem łańcuchem bloków. Każda z metod jest wykonywana w ramach opisanych wcześniej akcji lub endpointu. W konstruktorze klasy wykonywana jest metoda *createGenesisBlock*, która tworzy tak zwany blok inicjujący. Metody *getHistory* oraz *getBalance* pozwalają odpowiednio zwrócić listę transakcji z udziałem wskazanego adresu oraz jego saldo. Metoda *validateChain* przedstawiona na listingu 11 pozwala na sprawdzenia, czy łańcuch bloków jest prawidłowy. W tym celu sprawdzana jest poprawność transakcji i hashCode bloków. Metoda *addTransaction* dodaje transakcje do listy transakcji oczekujących. Niezbędne jest podanie adresu docelowego oraz adresu nadawcy. Ostatnią metodą klasy *Blockchain* jest *minePendingTransactions* przedstawiona na listingu 12, który przyjmuje jako parametr listę transakcji oczekujących. Tworzona jest przy tym nowa transakcja, która będzie nagrodą dla wskazanego adresu, nagroda jest dopisywana do listy transakcji oczekujących, po czym transakcje zostają przekazywane do nowo utworzonego bloku. Dla bloku wykonywana jest metoda *mine*, który z zadaną trudnością – zdefiniowaną w pliku konfiguracyjnym – dokonuje *mining* bloku. Po ukończeniu wydobywania blok zostaje dopisany do łańcucha bloków.

```

static validateChain(chain: Block[]): Boolean {
    for (let i = 1; i < chain.length - 1; i++) {
        const currentBlock = chain[i];
        const previousBlock = chain[i - 1];

        if (!Block.isValidTransactions(currentBlock.transactions)) {
            return false;
        }

        if (currentBlock.hash !== Block.calculateHash(currentBlock.timestamp, currentBlock.previousHash, currentBlock.transactions, currentBlock.nonce)) {
            return false;
        }
        if (currentBlock.previousHash !== previousBlock.hash) {
            return false;
        }
    }
    return true;
}

```

Listing 11 Kod metody sprawdzającej poprawność bloków

```

static minePendingTransaction(pendingTransactions: Transaction[], chain: Block[], rewardAddress: String): void {
    const rewardTx = new Transaction(+miningReward, '', rewardAddress);
    pendingTransactions.push(rewardTx);
    const block = new Block(new Date(), pendingTransactions, chain[chain.length - 1].hash);
    Block.mine(block, +mineDifficulty);
    chain.push(block);
}

```

Listing 12 Metoda wykonująca mining dla listy transakcji oczekujących

W klasie *Block* w konstruktorze przyjmowane są informacje, które posłużą do obliczenia wartości hashCode’u są to *timestamp* – data wykonania miningu, *transactions* – lista transakcji, które zostaną zapisane w bloku, *previousHash* – wartość hashCode poprzedniego bloku. Główną metodą klasy jest metoda *calculateHash*, która zwraca hashCode na podstawie następującej metody:

```
SHA256(previousHash + timestamp transactions + nonce)
```

Do funkcji hashującej SHA256 należącej do biblioteki *crypto-js* podawane są wartości tekstowe *hashCode* poprzedniego bloku, daty, listy transakcji wartości *nonce*. Funkcja na podstawie wartości zwraca wartość *hashCode* dla bloku. Sama funkcja wykonująca *mining* znajduje się w metodzie *mine* przedstawionej na listingu 13.

```
static mine(block: Block, diff: number): void {
    while (block.hash.substring(0, diff) !== Array(diff + 1).join("0")) {
        block.nonce++;
        block.hash = Block.calculateHash(block.timestamp, block.previousHash, block.transactions, block.nonce);
    }
    console.log("Block mined " + block.hash);
}
```

Listing 13 Metoda *mine* wykonująca mining bloku

Parametr *diff* definiuje trudność miningu, jest on podawany przez plik konfiguracyjny. Przez trudność miningu można dostosować długość wykonywania obliczeń. Algorytm działa dopóki ilość zer z prawej podana jako parametr *diff* będzie znajdować się w obliczanym *hashCode*. Metoda *hashValidTransactions* poddaje sprawdzeniu transakcji zawartych w bloku.

Klasa *Transaction* przyjmuje w konstruktorze kwotę jaka ma być przesłana, a także adres nadawcy i adres odbiorcy. W klasie występuje metoda *signTransaction* służy do podpisywania transakcji na podstawie podanego klucza publicznego i prywatnego. Przy podpisaniu transakcji podobnie jak w przypadku tworzenia bloku obliczana jest funkcja hashująca, która następnie jest poddawana kodowaniu base64. Metoda *isValid* sprawdza, czy dana transakcja jest podpisana oraz weryfikowana, czy wartości kluczy są zgodne z tymi, które zostały wykorzystane do utworzenia podpisu.

W klasie *Wallet* występują przydatne metody do pobierania zarówno klucza prywatnego i publicznego należącego do peera. Służą do identyfikowania właściciela danej transakcji.

4. Testowanie aplikacji

Do uruchomienia aplikacji wymagane jest środowisko Node.js oraz instalacja bibliotek. Do instalacji niezbędnych bibliotek należy wykonać komendę *npm install* w katalogu client oraz w katalogu server – obie aplikacje posiadają osobny projekt i osobną listę dependencji opisanych w pliku *package.json*. Do testowania przygotowano pliki wykonywalne w katalogu config, które zawierają osobną konfigurację dla danej aplikacji. Konfiguracja zostaje zdefiniowana przez zmienne środowiskowe, które ustawiane są przez komendę *SET*.

SET DIFFICULTY = 3 – trudność miningu

SET REWARD = 10 – nagroda za wykonanie miningu

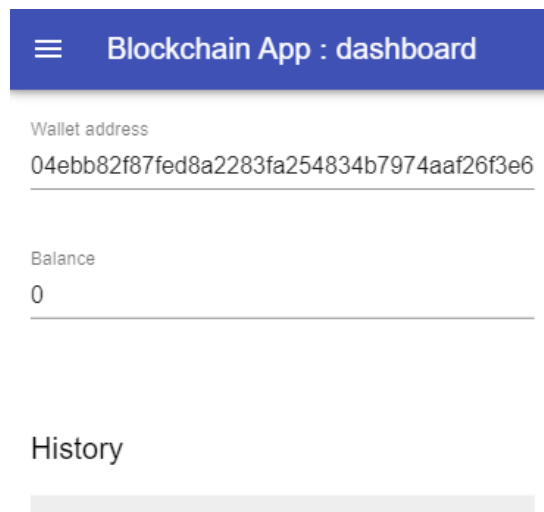
SET PEERS = ws://localhost:8903, – adresy peerów oddzielone przecinkiem

SET P2P_PORT – port p2p

SET PORT – port API

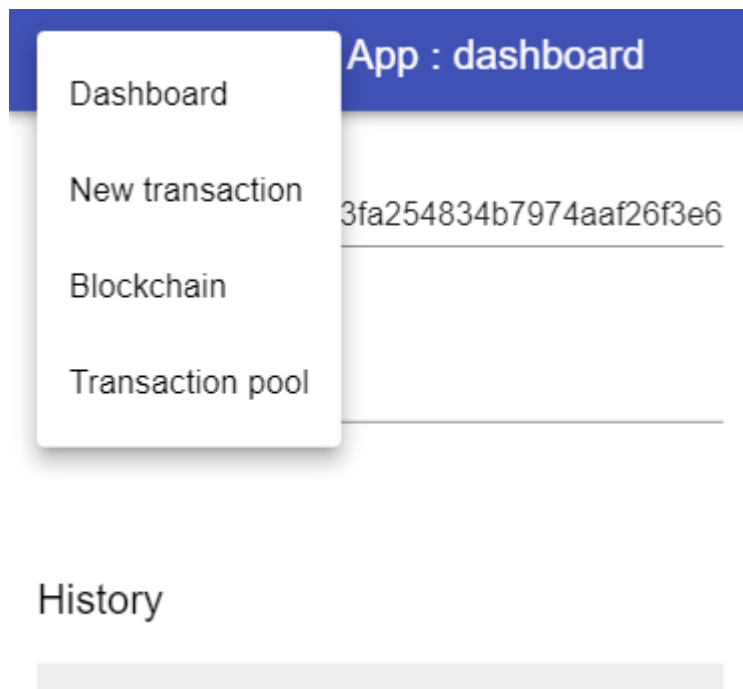
SET REACT_APP_HTTP_PORT – port klienta

Do prawidłowego działania sieci nie ma znaczenia, czy uruchomiono jednego klienta, lub więcej. Sieć rozproszona blockchain ma to do siebie, że może działać z dowolną liczbą peerów.



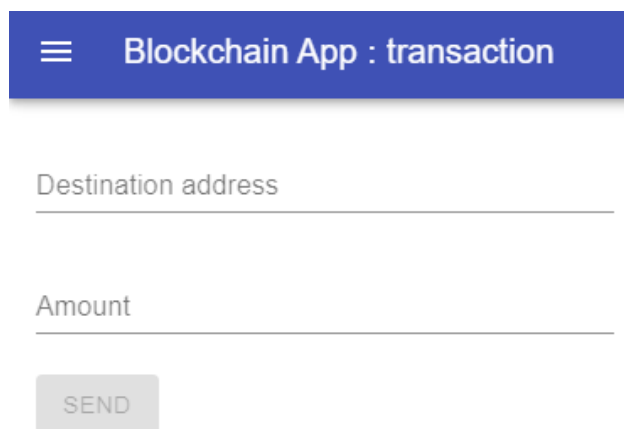
Rysunek 2 Widok dashboardu

Aplikacja po uruchomieniu pokazuje widok dashboardu, gdzie użytkownik otrzymuje informacje o wygenerowanym adresie klucza publicznego oraz stanie konta i historii ewentualnych transakcji. Po kliknięciu w menu znajdujące się w nagłówku pokazana zostanie lista widoków w aplikacji, które można zmienić klikając w dany odnośnik.



Rysunek 3 Widok menu

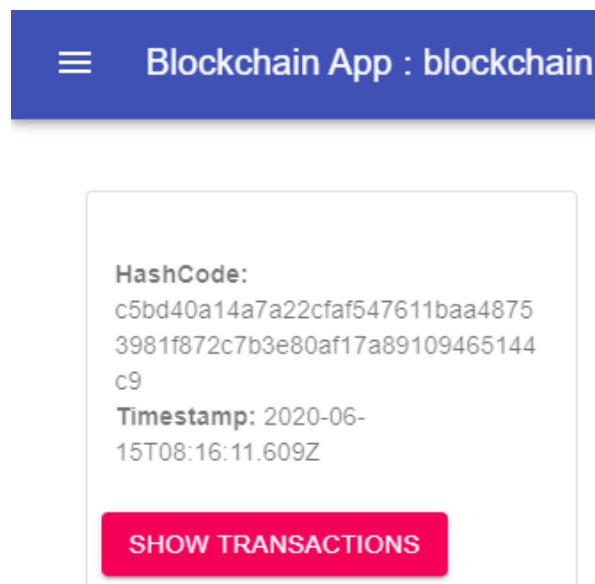
Po kliknięciu w odnośnik New transaction aplikacja pokazuje widok tworzenia transakcji, gdzie użytkownik może wpisać w pole tekstowe adres oraz kwotę i zatwierdzić ją przyciskiem send. Spowoduje to dopisanie transakcji to transakcji oczekujących na mining.



The image shows a mobile application interface for a blockchain transaction. At the top, there is a blue header bar with a white hamburger menu icon on the left and the text "Blockchain App : transaction" in white. Below the header, there are two input fields: "Destination address" and "Amount", each with a horizontal line underneath. At the bottom of the form, there is a grey rectangular button with the text "SEND" in black capital letters.

Rysunek 4 Widok transakcji

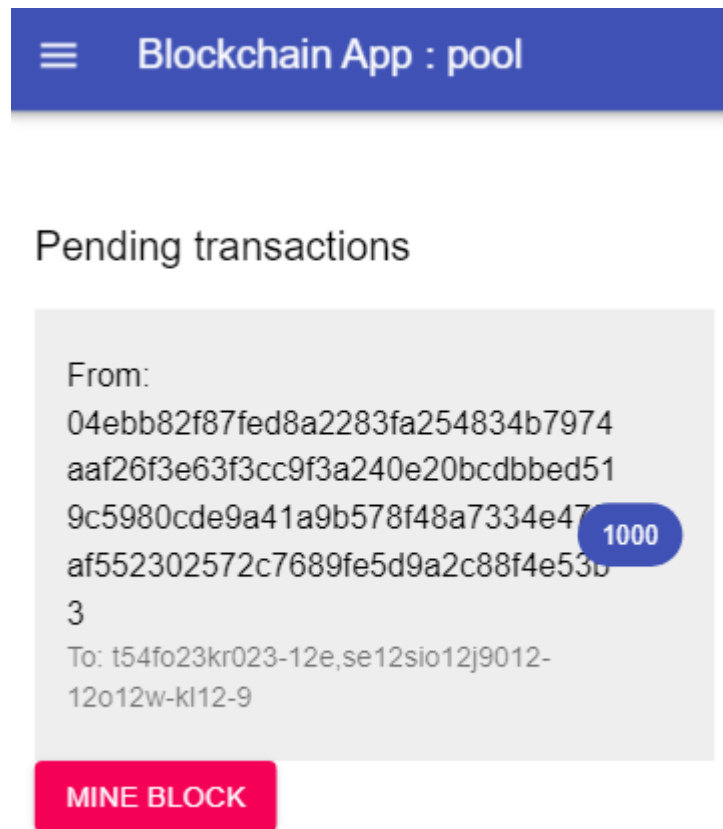
Po kliknięciu w odnośnik Blockchain pokazywana jest lista bloków z danymi jak identyfikator i data zatwierdzenia. Po kliknięciu w Show transactions w oknie modalnym ukazywane są transakcje zatwierdzone przez dany blok.



The image shows a mobile application interface for a blockchain block. At the top, there is a blue header bar with a white hamburger menu icon on the left and the text "Blockchain App : blockchain" in white. Below the header, there is a white rectangular box with a thin grey border. Inside this box, the text "HashCode:" is followed by a long alphanumeric string: "c5bd40a14a7a22cfaf547611baa48753981f872c7b3e80af17a89109465144c9". Below the hash code, the text "Timestamp:" is followed by the date and time "2020-06-15T08:16:11.609Z". At the bottom of the white box, there is a red rectangular button with the text "SHOW TRANSACTIONS" in white capital letters.

Rysunek 5 Lista bloków

Ostatni widok przedstawia listę transakcji niezatwierdzonych. Poniżej jest przycisk wykonania miningu, który po kliknięciu przycisku wykorzystuje moc obliczeniową peerów w celu zatwierdzenia transakcji.



Rysunek 6 Lista transakcji niezatwierdzonych

Testy przeprowadzono w oparciu o daną liczbę podłączonych peer'ów. Każdy peer posiada inną konfigurację trudności miningu. Przewidziano następujące scenariusze symulacyjne:

1. Peer-N przesyła na adres Peer'a-N+1 daną kwotę, Peer-N+2 dokonuje zatwierdzenia transakcji, sprawdzana jest jego poprawność.
2. Peer-N przesyła na adres Peer'a-N+1 daną kwotę, Peer-N+5 dokonuje zatwierdzenia transakcji, w międzyczasie Peer-N+1 dodaje inną transakcję, którą zatwierdza Peer-N+2.
3. Peer-N przesyła na adres Peer'a-N+1 daną kwotę, Peer-N+2 dokonuje zatwierdzenia transakcji, sprawdzana jest jego poprawność. Peer-N+3

dołącza do sieci – sprawdzana jest synchronizacja nowych bloków z resztą sieci.

4. Wszystkie 7 peerów dołącza do sieci, każdy dokonuje transakcji na adres sąsiedniego peera, wybrany peer zatwierdza transakcje. Sprawdzana jest poprawność przeprowadzanych transakcji przez każdego peera.
5. (Symulacja ataku 51%) 7 peerów dołącza do sieci, 1 peer wchodzi w tryb offline. 6 peerów w trybie online wykonuje transakcje i je zatwierdza tworząc łańcuch o długości 5 bloków. 1 fałszywy peer dodaje własne 5 bloków i przed dodaniem 6 wchodzi w tryb online publikując swoje rozgałęzienie. Jako, że łańcuch fałszywego peera jest dłuższy od łańcucha innych peerów to zostaje on łańcuchem obowiązującym.

Scenariusz 1

Peer	1	2	3	4	5	6	7
Trudność	3	4	5	6	7	8	9

W teście nowa transakcja jest zatwierdzona przez sieć prawidłowo co potwierdza zrzut ekranu na rys 7.

Wallet address

04e7005bf92d3648051c79bc28a7ddaa0a803322234fc3bb99494f5e94ec4c858c12260557b0c328515c84e7ac45d26ac9bbcd5b

Balance

110

History

045a5cbdb0a88cf6d52e3335a374406c4bf04d64565a2f2a9af433309a14910a4ef2af9d19596e01844f115157fb122686
eeb274687bebd712808670d1335f6b

04e7005bf92d3648051c79bc28a7ddaa0a803322234fc3bb99494f5e94ec4c858c12260557b0c328515c84e7ac45d26ac9bbcd5b1f34
6dcf3c8aa8a0d29a21ccb

+100

04e7005bf92d3648051c79bc28a7ddaa0a803322234fc3bb99494f5e94ec4c858c12260557b0c328515c84e7ac45d26ac9bbcd5b1f34
6dcf3c8aa8a0d29a21ccb

+10

Rysunek 7 Historia transakcji

Sposób działania sieci zobrazowano na rys. 8. Blok inicjujący jest taki sam dla dwóch peerów, gdy jeden z peerów wykona transakcje zostaje ona rozgłoszona do całej sieci. Wtedy inny peer może zatwierdzić transakcję. Po wykonaniu obliczeń nowo utworzony blok zostaje dopisany do łańcucha.

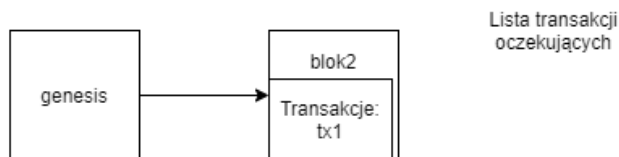
Przed dodaniem transakcji



Po dodaniu transakcji



Po wykonaniu zatwierdzenia przez Peer#2



Rysunek 8 Schemat dodawania transakcji w scenariuszu 1

Scenariusz 2

Peer	1	2	3	4	5	6	7
Trudność	3	4	5	6	7	8	9

Po wykonaniu transakcji przez Peer-N następuje wykonanie zatwierdzenia transakcji w procesie miningu. Jako, że trudność miningu dla Peer-N+5 jest większa to trwa to znacznie dłużej, niż w przypadku transakcji wykonywanej przez Peer-N+2. W trakcie wykonywania miningu po przez Peer-N+5 transakcja nie jest usuwana z listy, więc dodanie kolejnej transakcji i wykonanie potwierdzenia spowoduje dodanie bloku z dwiema transakcjami. Gdy Peer-N+5 ukończy mining, nowy blok zostanie rozesłany. Pozostałe peery jednak nie

akceptują łańcucha, gdyż nie jest on dłuższy od pozostałych. W efekcie blok jest odrzucany przez sieć. Przedstawiony schemat działania przedstawia rysunek 9.



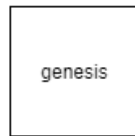
Rysunek 9 Schemat obrazuje działanie scenariusza 2

Scenariusz 3

Peer	1	2	3	4	5	6	7
Trudność	3	4	5	6	7	8	9

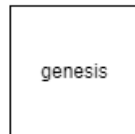
Scenariusz 3 przypomina scenariusz 1 pod względem wykonywanych operacji. Po dołączeniu przez Peer#N+3 zostanie dokonana synchronizacja łańcucha. Wszystkie zatwierdzone bloki zostaną pobrane do peer'a, mimo, że peer nie działał w trakcie ich zatwierdzania.

Przed dodaniem transakcji



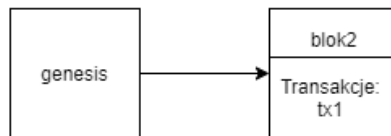
Lista transakcji
oczekujących

Po dodaniu transakcji



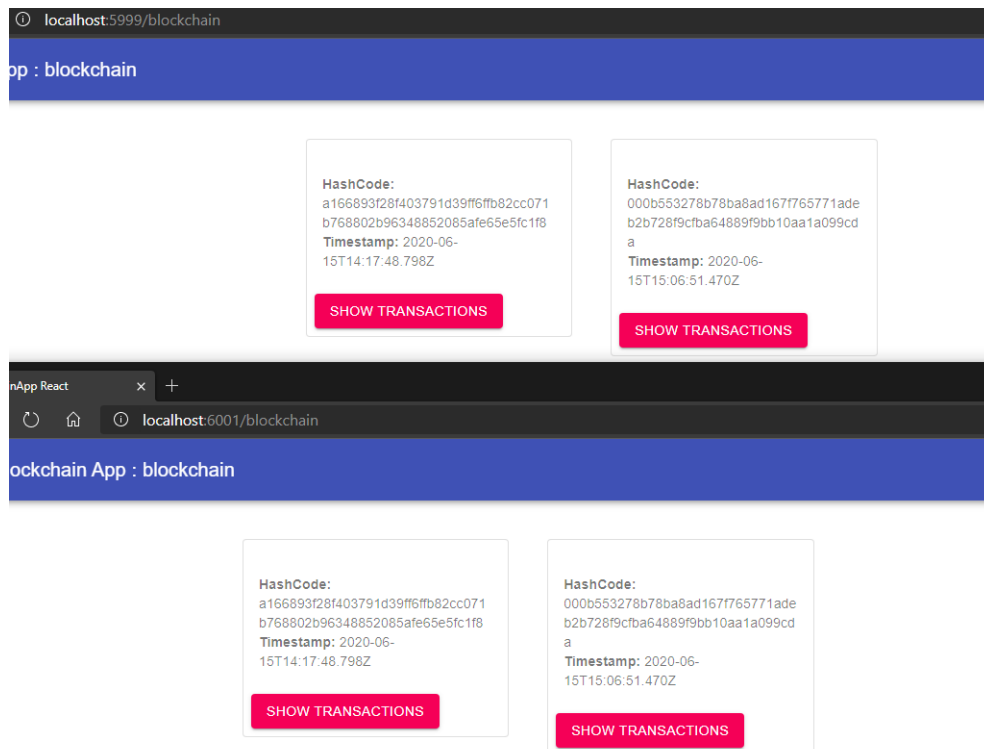
Lista transakcji
oczekujących
tx1 (Peer#1)

Po wykonaniu zatwierdzenia przez Peer#2



Lista transakcji
oczekujących

Rysunek 10 Schemat 10 obrazuje działanie scenariusza 3



Rysunek 11 Porównanie łańcuchów bloków Peera#N oraz Peera#N+2

Scenariusz 4

Peer	1	2	3	4	5	6	7
Trudność	3	4	5	6	7	8	9

W scenariuszu 4 jeden z peerów dokonuje minigu transakcji oczekujących przedstawionych na rysunku 12 dokonanych przez wszystkich peer'ów. Po prawidłowym zatwierdzeniu bloku wybierając jeden z peerów np. Peer#N+2 w historii widnieje dana transakcja, co przedstawiono na rysunku 13. Każda transakcja została więc prawidłowo zsynchronizowana.

Pending transactions

From:
04a8808e588d5ae401016c9c37f23ae9cd72f3e0b15b24dbe19da662f35424fc563a587e22edab191b0c96b139221893a7c92de52eeb869045c753e1b5a5db3f53

To:
04602e28ce0cdadd8a79027781b64838da25e5b89161763cc1f8a3b40659f144822e6891c3b84ab55811ef24e9513beb94beb426dda88da1966700ceb48a5fefa4

10

From:
04602e28ce0cdadd8a79027781b64838da25e5b89161763cc1f8a3b40659f144822e6891c3b84ab55811ef24e9513beb94beb426dda88da1966700ceb48a5fefa4

To:
04a5ad406b6eb4c86b25fff63af87690eaedecee2cc00cdec3ab21b9f08da38c4abf7e561fa6dcb8df2707148529efe0b772257382f42ace2b3d0ba1e43b19f53f

20

From:
04a5ad406b6eb4c86b25fff63af87690eaedecee2cc00cdec3ab21b9f08da38c4abf7e561fa6dcb8df2707148529efe0b772257382f42ace2b3d0ba1e43b19f53f

To:
04772179ff6b3cad0de5e63e80d61cd6bb5e4a23eee57e2c619c6c9b01457af79cebdd47a8c2b708971a972f2fae77e6ea2c8ae8c832565169015a15fb0853b78

30

From:
04772179ff6b3cad0de5e63e80d61cd6bb5e4a23eee57e2c619c6c9b01457af79cebdd47a8c2b708971a972f2fae77e6ea2c8ae8c832565169015a15fb0853b78

To:
04b766420c58a9983c1a2c4ca914354be37a2cb3655a073201647a841dd8b5e36762a2d72755dfb36fed931914578df3a4ab9d96b66fc934a46117ec00db6027ea

40

Rysunek 12 Lista transakcji oczekujących

History

04602e28ce0cdadd8a79027781b64838da25e5b89161763cc1f8a3b40659f144822e6891c3b84ab55811ef24e9513be
b94beb426dda88da1966700ceb48a5fefa4

04a5ad406b6eb4c86b25fff63af87690eaedecee2cc00cdec3ab21b9f08da38c4abf7e561fa6dcb8df2707148529efe0b772257382f42ace
2b3d0ba1e43b19f53f

+20

04a5ad406b6eb4c86b25fff63af87690eaedecee2cc00cdec3ab21b9f08da38c4abf7e561fa6dcb8df2707148529efe0b77
2257382f42ace2b3d0ba1e43b19f53f

04772179ff6b3cad0de5e63e80d61cd6bb5e4a23eee57e2c619c6c9b01457af79cebddd47a8c2b708971a972f2fae77e6ea2c8ae6c8323
65169015a15fb0853b78

-30

Rysunek 13 Historia transakcji dla Peera#N+2

Scenariusz 5

Peer	1	2	3	4	5	6	7
Trudność	3	4	5	6	7	8	9

W scenariuszu 5 wszystkie peery dokonują transakcji i je zatwierdzają. Po tym Peer#N+6 dodaje własne transakcje, które zatwierdza w trybie offline tak jak na rysunku 14 tworząc odgałęzienie. Gdy po tym Peer#N+6 doda kolejny blok w trybie online, cały łańcuch zostanie rozesłany do sieci. Z racji, że będzie on przewyższał długością łańcuch główny, nowe peery go zsynchronizują i będą traktować go jako łańcuch główny. Jest to symulacja ataku 51% w którym peer lub grupa peerów posiada większość mocy sieci, co przekłada się na przewagę przy zatwierdzaniu nowych transakcji. Szczegółowy schemat działania przedstawia rysunek 15.

Pending transactions

From:

04fa64395597742d660ffd43cc59063f6e42e88939a0fc1409f1c0a85db9d584a5b624bd306319f58522a2de0b670784e
d8b37cd6c5fbda3313b4861f640089619

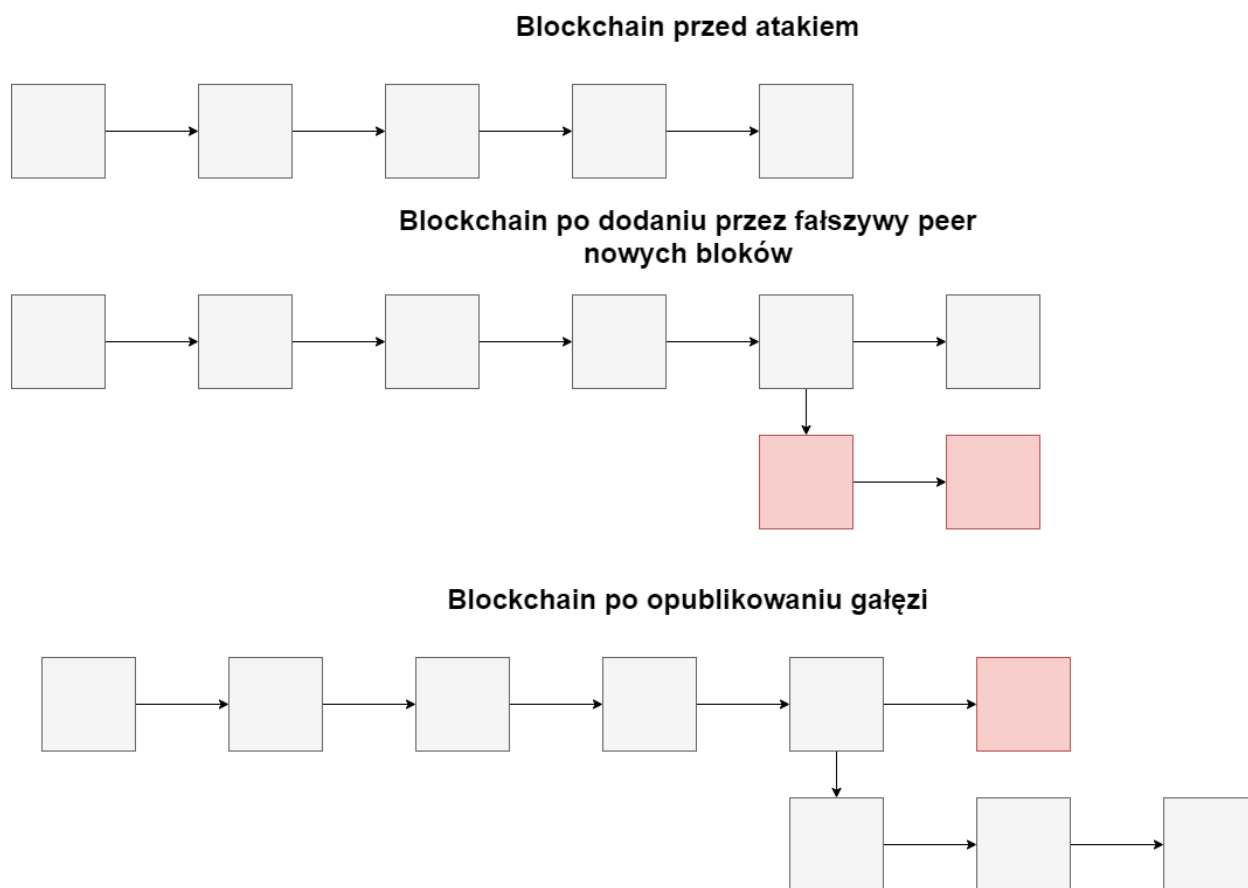
To: 44

3

MINE BLOCK

Online: ☐

Rysunek 14 Mining w trybie offline



Rysunek 15 Schemat działania scenariusza 5

5. Wnioski

Celem projektu był system oparty na sieci blockchain realizujący odporność na błędy. Aplikacja została zrealizowana w architekturze klient-serwer, gdzie aplikacja klienta została zaimplementowana w React.js, a aplikacja serwera przy użyciu środowiska Node.js. Aplikacja realizuje uproszczony model działania sieci blockchain z wykorzystaniem dowolnej liczby peerów. Dzięki rozproszonej architekturze i implementacji założeń, aplikacja pozwala utrzymać stabilną sieć blockchain bez względu na ilość działających peerów. Scenariusz 1 pozwolił na przetestowanie, czy bloki są prawidłowo rozsyłane między peer'ami. Scenariusz 2 zasymulował poprawność dołączania kolejnych bloków. Każdy blok musi być związany wartością hashującą, inaczej zostanie odrzucony przez sieć przez walidację bloków i transakcji. Scenariusz 3 pokazał możliwość synchronizacji bloków, gdy dany peer nie był

podłączony do pozostałych peerów w sieci. Scenariusz 4 przetestował synchronizację na podstawie wszystkich działających peerów w sieci. W ostatnim scenariuszu przetestowano atak 51%. Uznano, że fałszywy peer przewyższa mocą obliczeniową resztę sieci i jest w stanie szybciej generować nowe bloki po czym przejąć kontrolę nad siecią. Aby sieć była odporna na tego typu ataki należy brać uwagę timestamp, czyli odrzucać opóźnione bloki, które mogłyby być wykonywane, gdy peer jest w trybie offline.

Aplikacja może być rozbudowana pod kątem wykonywania obliczeń rozproszonych, co pozwoli na niemal identyczną zbieżność z działaniem rzeczywistego protokołu blockchaina, który poprawność łańcucha bloków zawdzięcza uczciwej większości użytkowników.