Tworzenie własnego łańcucha bloków

W tej części omówię, jak zbudować własną sieć blockchain P2P. Jest to proces składający się z siedmiu kroków, więc w każdej sekcji zacznę od krótkiego wprowadzenia, po którym nastąpi ćwiczenie.

- Tworzenie podstawowej sieci P2P
- Wysyłanie i odbieranie bloków
- Rejestrowanie górników i tworzenie nowych bloków
- Utworzenie bazy danych nazwa-wartość, LevelDB
- Tworzenie portfela prywatno-publicznego
- Korzystanie z usług API
- Tworzenie interfejsu wiersza poleceń

W tej części zagłębimy się w kod, a przykłady w tej części są proste z natury i przeznaczone do celów edukacyjnych. Dzięki nim lepiej zrozumiesz blockchain i elementy potrzebne do osiągnięcia w pełni działającego prototypu blockchain.

Uwaga: w tym krótkim rozdziale instruktażowym nie jest możliwe stworzenie pełnego łańcucha bloków klasy produkcyjnej; jednak dam ci podstawy do stworzenia podstawowego działającego.

Tworzenie podstawowej sieci P2P

Pierwszym krokiem w tworzeniu łańcucha bloków jest stworzenie sieci P2P. Jak widzieliście w poprzednich rozdziałach, sieć P2P była kluczem do działania blockchain. W kryptowalutach sieć P2P może pomóc w zapobieganiu problemowi podwójnych wydatków na PoW, a także stanowi podstawową architekturę PoS. W łańcuchu bloków umożliwia synchronizację dowolnych danych potrzebnych w sieci. Uwaga Peer-to-peer (P2P) to rodzaj sieci komputerowej wykorzystującej architekturę rozproszoną. Każdy węzeł równorzędny lub węzeł współdzieli obciążenie i jest równy pozostałym równorzędnym, co oznacza, że nie powinno być żadnego uprzywilejowanego węzła równorzędnego.

„Zaproponowaliśmy system do transakcji elektronicznych bez polegania na zaufaniu. Zaczęliśmy od zwykłych ram monet wykonanych z podpisów cyfrowych, które zapewniają silną kontrolę własności, ale są niekompletne bez możliwości zapobiegania podwójnym wydatkom. Aby rozwiązać ten problem, zaproponowaliśmy sieć peer-topeer wykorzystującą dowód pracy do rejestrowania publicznej historii transakcji, która szybko staje się niepraktyczna obliczeniowo, aby atakujący mógł się zmienić, jeśli uczciwe węzły kontrolują większość sprawdzania pracy procesora. -Bitcoin: elektroniczny system gotówkowy peer-to-peer

W tej części pokażę Ci, jak stworzyć swój blockchain za pomocą Node.js, ale możesz to zrobić z dowolnym innym językiem programowania, ponieważ zasady są takie same. Będziesz konfigurować swój komputer ze zintegrowanym środowiskiem programistycznym (IDE) WebStorm, które będzie używane. Aby pobrać WebStorm, przejdź do https://www.jetbrains.com/webstorm/. WebStorm oferuje 30-dniowy okres próbny; jednak nie jest to konieczne i możesz wybrać dowolne IDE, które Ci się podoba, i osiągnąć te same wyniki.

KROK 1: PODSTAWOWE ĆWICZENIE Z SIECIĄ P2P

Konfiguracja projektu

W tym ćwiczeniu skonfigurujesz projekt i utworzysz podstawową sieć P2P do wysyłania i odbierania wiadomości. Gdy będziesz w stanie wysyłać i odbierać wiadomości, będziesz mógł utworzyć klasę bloków i bibliotekę łańcuchową oraz powiązać kilka bloków, aby utworzyć łańcuch bloków. Będziesz potrzebował zainstalowanego Node.js na swoim komputerze; można go zainstalować na wiele sposobów. Jednym prostym sposobem jest skorzystanie z gotowego menedżera instalatora; znajdź taki, który pasuje do Twojej platformy tutaj: https://nodejs.org/en/download/. Po pobraniu WebStorm możesz utworzyć nowy projekt. Wybierz Plik ➤ Utwórz nowy projekt ➤ Aplikacja Node.js Express ➤ UTWÓRZ E. W lokalizacji wywołaj projekt Blockchain i kliknij Utwórz.

Tworzenie sieci P2P

Utwórz folder i nazwij go Blockchain. Następnie utwórz plik i nazwij go p2p.js i napisz następujący kod.

```
const crypto = require('crypto'),
Swarm = require('discovery-swarm'),
defaults = require('dat-swarm-defaults'),
getPort = require('get-port');
const peers = {};
let connSeq = 0;
let channel = 'myBlockchain';
const myPeerId = crypto.randomBytes(32);
console.log('myPeerId: ' + myPeerId.toString('hex'));
const config = defaults({
id: myPeerId,
});
const swarm = Swarm(config);
(async () => {
const port = await getPort();
swarm.listen(port);
console.log('Listening port: ' + port);
swarm.join(channel);
swarm.on('connection', (conn, info) => {
const seq = connSeq;
const peerId = info.id.toString('hex');
console.log(`Connected #${seq} to peer: ${peerId}`);
if (info.initiator) {
```

```
try {
conn.setKeepAlive(true, 600);
} catch (exception) {
console.log('exception', exception);
}
}
conn.on('data', data => {
let message = JSON.parse(data);
console.log('----------- Received Message start ----
---------');
console.log(
'from: ' + peerId.toString('hex'),
'to: ' + peerId.toString(message.to),
'my: ' + myPeerId.toString('hex'),
'type: ' + JSON.stringify(message.type)
);
console.log('----------- Received Message end -----
--------');
});
conn.on('close', () => {
console.log(`Connection ${seq} closed, peerId:
${peerId}`);
if (peers[peerId].seq === seq) {
delete peers[peerId]
}
});
if (!peers[peerId]) {
peers[peerId] = {}
}
peers[peerId].conn = conn;
peers[peerId].seq = seq;
```

```
connSeq++
})
})();
setTimeout(function(){
writeMessageToPeers('hello', null);
}, 10000);
writeMessageToPeers = (type, data) => {
for (let id in peers) {
console.log('-------- writeMessageToPeers start -------- ');
console.log('type: ' + type + ', to: ' + id);
console.log('-------- writeMessageToPeers end ----------- ');
sendMessage(id, type, data);
}
};
writeMessageToPeerToId = (toId, type, data) => {
for (let id in peers) {
if (id === toId) {
console.log('-------- writeMessageToPeerToId start
-------- ');
console.log('type: ' + type + ', to: ' + toId);
console.log('-------- writeMessageToPeerToId end ---
-------- ');
sendMessage(id, type, data);
}
}
};
sendMessage = (id, type, data) => {
peers[id].conn.write(JSON.stringify(
{
to: id,
from: myPeerId,
```

type: type,

data: data

}

}));

};

Aby ten przykład zadziałał, musisz uruchomić dwa wystąpienia tego kodu. Możesz uruchomić go z dwóch oddzielnych komputerów, tak jak w prawdziwym życiu, lub możesz uruchomić dwie instancje z tej samej maszyny za pośrednictwem Terminala. Twój kod musi znajdować i łączyć się z elementami równorzędnymi, wdrażać serwery używane do wykrywania innych elementów równorzędnych i uzyskiwać dostępny port TCP. Odbywa się to za pomocą tych trzech bibliotek:

- Discovery-swarm: Służy do tworzenia roju sieciowego, który używa Discovery-channel do wyszukiwania i łączenia peerów

- dat-swarm-defaults: Wdraża serwery używane do wykrywania innych peerów

- get-port: Pobiera dostępne porty TCP

Aby zainstalować te biblioteki, uruchom to polecenie:

> npm install crypto Discovery-swarm dat-swarm-defaults get-port

--save

Teraz, gdy biblioteki są już zainstalowane, otwórz dwie instancje Terminala i przejdź do lokalizacji biblioteki. Uruchom następujące polecenie:

> node p2p.js

Aby uruchomić kod z biblioteki klonów na GitHub, przejdź do kodu, wykonaj te polecenia terminala, aby zainstalować biblioteki, i uruchom instancję node.js dołączając nasz kod p2p.js:

> cd [location]/chapter2/step2

> npm install

> node p2p.js

Rysunek przedstawia wynik działania kodu Node.js.

Jak widać na rysunku , sieć wygenerowała losowy identyfikator peera dla twojego komputera i wybrała losowy port, korzystając z zainstalowanych bibliotek wykrywania. Następnie kod był w stanie wykryć innych partnerów w sieci oraz wysyłać i odbierać wiadomości do i od tych partnerów. Jesteś teraz

połączony w sieci P2P z innymi użytkownikami. Przejdźmy przez kod, aby lepiej zrozumieć, jak to wszystko działa. Pierwsze wiersze kodu to instrukcja importu dla bibliotek typu open source, których używasz w swoim kodzie.

```
const crypto = require('crypto'),

Swarm = require('discovery-swarm'),

defaults = require('dat-swarm-defaults'),

getPort = require('get-port');
```

Zauważ, że używasz const do ustawienia zmiennej zamiast let. Chcesz mieć pewność, że nie nastąpi ponowne powiązanie i zawsze odwołujesz się do tego samego obiektu, więc wybór const jest zalecany zgodnie z najlepszymi praktykami. Następnie ustawiasz zmienne tak, aby przechowywały obiekt z peerami i sekwencją połączeń, i wybierasz nazwę kanału, z którym będą się łączyć wszystkie twoje węzły. Ustawiasz również losowo generowany identyfikator peera dla swojego partnera, korzystając z biblioteki kryptograficznej.

```
const peers = {};

let connSeq = 0;

let channel = 'myBlockchain';

const myPeerId = crypto.randomBytes(32);

console.log('myPeerId: ' + myPeerId.toString('hex'));
```

Następnie generujesz obiekt konfiguracyjny, który przechowuje Twój identyfikator równorzędny. Następnie używasz obiektu config do inicjalizacji biblioteki roju. Bibliotekę roju można znaleźć tutaj: https://github.com/mafintosh/discovery-swarm. To, co robi, to tworzenie roju sieciowego, który wykorzystuje bibliotekę Discovery-Channel do wyszukiwania i łączenia równorzędnych partnerów w sieci UCP/TCP.

```
const config = defaults({

id: myPeerId,

});

const swarm = Swarm(config);
```

Teraz, gdy wszystko jest już skonfigurowane i gotowe, utworzysz funkcję asynchroniczną Node.js do ciągłego monitorowania komunikatów o zdarzeniach swarm.on.

```
(async () => {
```

Nasłuchujesz na wybranym losowym porcie, a po nawiązaniu połączenia z peerem używasz setKeepAlive, aby upewnić się, że połączenie sieciowe pozostanie z innymi peerami.

```
const port = await getPort();

swarm.listen(port);

console.log('Listening port: ' + port);

swarm.join(channel);
```

```
swarm.on('connection', (conn, info) => {

const seq = connSeq;

const peerId = info.id.toString('hex');

console.log(`Connected #${seq} to peer: ${peerId}`);

if (info.initiator) {

try {

conn.setKeepAlive(true, 600);

} catch (exception) {

console.log('exception', exception);

}

}
```

Po otrzymaniu wiadomości z danymi w sieci P2P analizujesz dane za pomocą JSON.parse, które jest natywnym poleceniem Node.js, więc nie musisz dołączać żadnej instrukcji importu. To polecenie dekoduje wiadomość z powrotem do obiektu, a polecenie toString konwertuje bajty na czytelny typ danych ciągu.

```
conn.on('data', data => {

let message = JSON.parse(data);

console.log('----------- Received Message start ----

---------');

console.log(

'from: ' + peerId.toString('hex'),

'to: ' + peerId.toString(message.to),

'my: ' + myPeerId.toString('hex'),

'type: ' + JSON.stringify(message.type)

);

console.log('----------- Received Message end -----

--------');

});
```

Nasłuchujesz również zdarzenia zamknięcia, które wskaże, że utraciłeś połączenie z peerami, dzięki czemu możesz podjąć działania, takie jak usunięcie peerów z obiektu listy peerów.

```
conn.on('close', () => {

console.log(`Connection ${seq} closed, peerId: ${peerId}`);
```

```
if (peers[peerId].seq === seq) {

delete peers[peerId]

}

});

if (!peers[peerId]) {

peers[peerId] = {}

}

peers[peerId].conn = conn;

peers[peerId].seq = seq;

connSeq++

})

})();
```

Tutaj będziesz używać natywnej funkcji setTimeout Node.js do wysyłania wiadomości po dziesięciu sekundach do wszystkich dostępnych peerów. Pierwsza wiadomość, którą wyślesz, to po prostu wiadomość „cześć”. Tworzysz metody o nazwie writeMessageToPeers i writeMessageToPeerToId do obsługi obiektu, więc jest on sformatowany z danymi, które chcesz przesłać i do kogo chcesz je wysłać.

```
setTimeout(function(){

writeMessageToPeers('hello', null);

}, 10000);
```

Metoda writeMessageToPeers będzie wysyłać wiadomości do wszystkich połączonych peerów.

```
writeMessageToPeers = (type, data) => {

for (let id in peers) {

console.log('-------- writeMessageToPeers start ------

-- ');

console.log('type: ' + type + ', to: ' + id);

console.log('-------- writeMessageToPeers end ---------

-- ');

sendMessage(id, type, data);

}

};
```

Dodatkowo utworzysz kolejną metodę writeMessageToPeerToId, która będzie wysyłała wiadomość do określonego identyfikatora peera, na wypadek gdybyś chciał komunikować się tylko z jednym określonym peerem.

}

```
writeMessageToPeerToId = (toId, type, data) => {
for (let id in peers) {
if (id === toId) {
console.log('-------- writeMessageToPeerToId start
-------- ');
console.log('type: ' + type + ', to: ' + toId);
console.log('-------- writeMessageToPeerToId end ---
-------- ');
sendMessage(id, type, data);
}
}
};
```

Na koniec sendMessage to ogólna metoda, której będziesz używać do wysyłania wiadomości sformatowanej za pomocą parametrów, które chcesz przekazać, i zawiera następujące elementy:

- do/od: identyfikator peera, z którego i do którego wysyłasz wiadomość
- typ: typ wiadomości
- dane: dowolne dane, które chcesz udostępnić w sieci P2P

Te parametry przydadzą się, gdy udostępnisz blok łańcucha bloków. Zwróć uwagę, że przekazywana wiadomość musi być ciągiem i nie może być obiektem, więc używasz natywnej funkcji JSON.stringify do kodowania wiadomości przed udostępnieniem ich w sieci P2P.

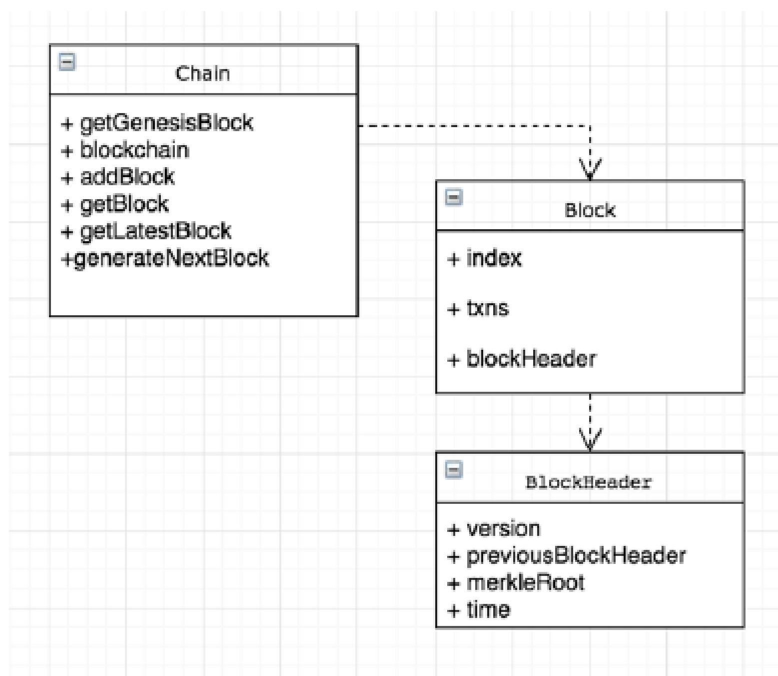
```
sendMessage = (id, type, data) => {
peers[id].conn.write(JSON.stringify(
{
to: id,
from: myPeerId,
type: type,
data: data
}
));
};
```

W tym ćwiczeniu pobrałeś i zainstalowałeś WebStorm IDE oraz utworzyłeś projekt, który obejmuje podstawową sieć P2P. Byłeś w stanie utrzymać połączenie z losowym portem sieci TCP oraz wysyłać i

odbierać wiadomości, w tym kodować i dekodować te wiadomości. Jesteś gotowy, aby przejść do następnego ćwiczenia i wysłać rzeczywisty blok między każdym węzłem w Twojej sieci.

Tworzenie bloku Genesis i udostępnianie bloków

W następnym ćwiczeniu utworzysz obiekty blokowe, które będziesz udostępniać między węzłami. Ale zanim to zrobisz, przyjrzyjmy się bliżej obiektowi Block. Obiekt Block nie jest taki sam dla każdego łańcucha bloków. Różne łańcuchy bloków wykorzystują różne typy obiektów blokowych; będziesz używać obiektu Block podobnego do bitcoina; Omówiłem szczegółowo w części 2. Aby lepiej zrozumieć architekturę, spójrz na diagram w języku Unified Modeling Language (UML) obiektów Block i BlockHeader, których będziesz używać w następnym ćwiczeniu, jak pokazano na rysunku.



Przypominamy, że z Części 2 obiekt Blok zawiera następujące właściwości :

• index: GenesisBlock jest naszym pierwszym blokiem, indeksowi bloku przypisujemy wartość 0.

• txns: Jest to surowa transakcja w bloku. Nie chcę w tym rozdziale skupiać się tylko na kryptowalutach, więc pomyśl o tym jak o każdym rodzaju danych, które chcesz przechowywać.

Obiekt Block zawiera obiekt BlockHeader, który zawiera następujące właściwości:

• Wersja: Wersja 1 to blok genezy (2009) a wersja 2 to soft fork rdzenia bitcoin 0.7.0 (2012). Bloki wersji 3 były miękkim widelcem rdzenia bitcoin 0.10.0 (2015). Bloki w wersji 4 to BIP65 w rdzeniu bitcoin 0.11.2 (2015).

• Hash nagłówka poprzedniego bloku: Jest to funkcja skrótu SHA-256 (Secure Hash Algorithm) nagłówka poprzedniego bloku. Zapewnia to, że poprzedniego bloku nie można zmienić, ponieważ ten blok musi zmienić się również.

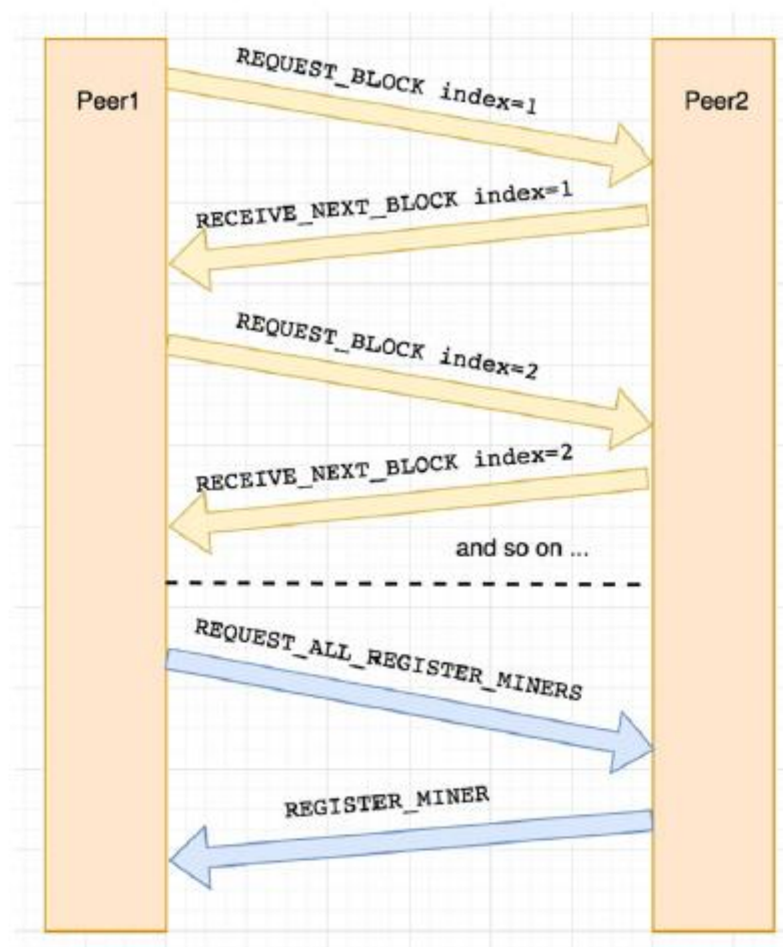
• Mieszanie korzenia Merkle: Drzewo Merkle to drzewo binarne, które zawiera wszystkie zahaszowane pary drzewa.

• Czas: Jest to czas epoki Uniksa, kiedy górnik zaczął haszować nagłówek. Jak pamiętasz, bitcoin zawiera również właściwość trudności dla górników, która jest przeliczana co 2016 bloki. Tutaj nie będziesz używać nBits i nounce params, ponieważ nie robisz PoW.

• nounce: Wartość jednorazowa w bloku bitcoin to 32-bitowe (4-bajtowe) pole, którego wartość jest dostosowywana przez górników tak, aby hash bloku był mniejszy lub równy aktualnemu celowi sieci.

• nBits: Odnosi się do celu. Celem jest 256-bitowa liczba i odwrotnie proporcjonalna do trudności. Jest przeliczany co 2016 bloki.

Jeśli chodzi o komunikację P2P, przepływ bloków między każdym równorzędnym w sieci P2P składa się z żądania najnowszego bloku od równorzędnego w sieci, a następnie otrzymania żądania bloku. Rysunek przedstawia schemat blokowy.



Teraz, gdy rozumiesz już architekturę i przepływ bloków w sieci P2P, w następnym ćwiczeniu będziesz wysyłać i żądać bloków.

KROK 2: ĆWICZENIE Z WYSYŁANIEM W SIECI P2P

Konfigurowanie klasy bloku i biblioteki łańcuchów

W tym ćwiczeniu utworzysz swój łańcuch bloków. Blockchain składa się z dwóch plików: block.js i chain.js. Plik Block.js będzie zawierał obiekt klasy bloku, a chain.js będzie klejem z metodami do obsługi interakcji z blokami. Jeśli chodzi o obiekt Block, będziesz tworzyć właściwości podobne do właściwości, które posiada rdzeń bitcoin. Spójrz na Listing 3-2, plik block.js zawiera obiekty Block i BlockHeader.

```
}
exports.BlockHeader = class BlockHeader {
constructor(version, previousBlockHeader, merkleRoot, time)
{
this.version = version;
this.previousBlockHeader = previousBlockHeader;
this.merkleRoot = merkleRoot;
this.time = time;
}
};
exports.Block = class Block {
constructor(blockHeader, index, txns) {
this.blockHeader = blockHeader;
this.index = index;
this.txns = txns;
}
}
```

Jak widać, chain.js zawiera pierwszy blok, który nazywa się blokiem genesis, a także metodę odbierania całego obiektu blockchain, dodawania bloku i pobierania bloku. Zauważ, że dodasz bibliotekę o nazwie moment, aby zaoszczędzić czas w uniksowym formacie czasu w swojej bibliotece chain.js. W tym celu zainstaluj moment z npm.

```
> npm install moment –save
```

Teraz, gdy masz już utworzony plik block.js, możesz utworzyć klasę chain.js;

```
let Block = require("./block.js").Block,
BlockHeader = require("./block.js").BlockHeader,
moment = require("moment");
let getGenesisBlock = () => {
let blockHeader = new BlockHeader(1, null, "0x1bc33000000000
000000000000000000000000000000000", moment().unix());
return new Block(blockHeader, 0, null);
};
let getLatestBlock = () => blockchain[blockchain.length-1];
let addBlock = (newBlock) => {
```

```
let prevBlock = getLatestBlock();
if (prevBlock.index < newBlock.index && newBlock.
blockHeader.previousBlockHeader === prevBlock.blockHeader.
merkleRoot) {
blockchain.push(newBlock);
}
}
let getBlock = (index) => {
if (blockchain.length-1 >= index)
return blockchain[index];
else
return null;
}
const blockchain = [getGenesisBlock()];
if (typeof exports != 'undefined' ) {
exports.addBlock = addBlock;
exports.getBlock = getBlock;
exports.blockchain = blockchain;
exports.getLatestBlock = getLatestBlock;
}
```

Masz teraz obiekt blokowy, który jest zawarty w chain.js. Twoja biblioteka może stworzyć blok genezy i dodać blok do obiektu łańcucha bloków. Będziesz także mógł wysyłać i prosić o bloki. Następnie w swojej klasie sieci P2P możesz użyć utworzonego pliku chain.js.

Najpierw musisz zaimportować klasę chain.js. v

```
const chain = require("./chain");
```

Następnie możesz zdefiniować typ wiadomości do żądania i otrzymania najnowszego bloku. Kiedy wysyłasz wiadomości w swojej sieci P2P, musisz być w stanie określić cel wiadomości. Korzystając z właściwości MessageType, można zdefiniować mechanizm przełączania, aby różne typy komunikatów były używane do różnych funkcji.

```
let MessageType = {
REQUEST_LATEST_BLOCK: 'requestLatestBlock',
LATEST_BLOCK: 'latestBlock'
};
```

Po odebraniu komunikatu o zdarzeniu danych połączenia możesz utworzyć swój kod przełącznika do obsługi różnych typów żądań, jak pokazano na Listingu.

```
switch (message.type) {
case MessageType.REQUEST_BLOCK:
console.log('-----------REQUEST_BLOCK-------------');
let requestedIndex = (JSON.parse(JSON.stringify(message.
data))).index;
let requestedBlock = chain.getBlock(requestedIndex);
if (requestedBlock)
writeMessageToPeerToId(peerId.toString('hex'),
MessageType.RECEIVE_NEXT_BLOCK, requestedBlock);
else
console.log('No block found @ index: ' + requestedIndex);
console.log('-----------REQUEST_BLOCK-------------');
break;
case MessageType.RECEIVE_NEXT_BLOCK:
console.log('-----------RECEIVE_NEXT_BLOCK-------------');
chain.addBlock(JSON.parse(JSON.stringify(message.data)));
console.log(JSON.stringify(chain.blockchain));
let nextBlockIndex = chain.getLatestBlock().index+1;
console.log('-- request next block @ index: ' +
nextBlockIndex);
writeMessageToPeers(MessageType.REQUEST_BLOCK, {index:
nextBlockIndex});
console.log('-----------RECEIVE_NEXT_BLOCK-------------');
break;
}
```

Na koniec ustawisz żądanie limitu czasu, które będzie wysyłać żądanie pobrania ostatniego bloku co 5000 milisekund (5 sekund).

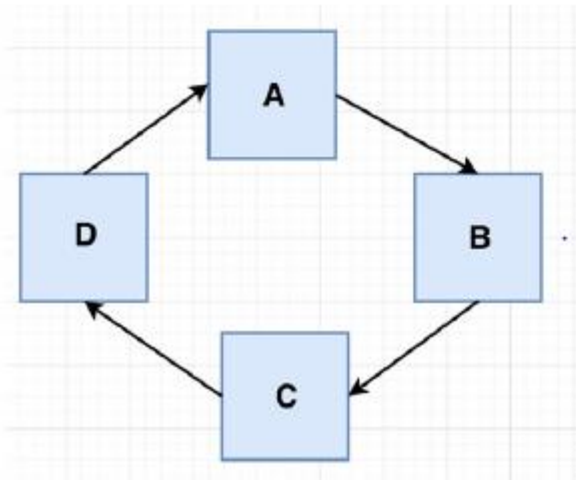
```
setTimeout(function(){
writeMessageToPeers(MessageType.REQUEST_BLOCK, {index: chain.
getLatestBlock().index+1});
```

}, 5000);

W tym ćwiczeniu udało Ci się wygenerować blok genezy i stworzyć mechanizm żądania i odbierania bloków poprzez wysyłanie żądań wiadomości. Możliwość żądania i odbierania bloków pozwala na synchronizację nowych peerów wchodzących do sieci P2P. Potrzebna jest również synchronizacja dla wszelkich dodatkowych bloków generowanych po utworzeniu bloku genesis.

Rejestrowanie górników i tworzenie nowych bloków

W tym momencie masz podstawową sieć P2P i możesz łączyć peerów w sieci, tworzyć blok genezy oraz blokować wysyłanie i odbieranie. Następnym krokiem jest możliwość generowania nowych bloków. Jak widzieliście w Części 2, dowód pracy opiera się na stworzeniu problemu matematycznego i nagradzaniu górników, którzy jako pierwsi znajdują rozwiązanie problemu. Jednak w tym przykładzie zastosujesz metodę dowodu stawki (PoS), w której ufasz każdemu górnikowi, aby wygenerował twoje bloki. Każdy peer zarejestruje się jako górnik i po kolei będzie wydobywał blok. Możesz zobaczyć przegląd każdego górnika generującego blok na rysunku.



Na koniec, zanim zaczniesz następne ćwiczenie, wróć do rysunku 3-4, aby lepiej zrozumieć swój przepływ. Przepływ pokazuje, jak sieć P2P obsługuje komunikację równorzędną, żądając najnowszego bloku i otrzymując najnowszy blok. W następnym ćwiczeniu zarejestrujesz swoich rówieśników jako górników i utworzysz nowe bloki.

KROK 3: REJESTRACJA GÓRNIKÓW I TWORZENIE NOWYCH BLOKÓW ĆWICZENIE

Zarejestruj górników

W tym ćwiczeniu zarejestrujesz górników i utworzysz nowe bloki. Aby zautomatyzować proces generowania bloku co x minut, możesz użyć biblioteki Node.js o nazwie cron, która jest podobna do biblioteki linuksowej automatyzującej zadania. Aby zainstalować bibliotekę cron open source, uruchom następujące polecenie:

> npm install cron –save

Następnie w pliku p2p.js utworzysz dwie zmienne, aby śledzić zarejestrowanych górników, a także kto wydobył ostatni blok, dzięki czemu możesz przypisać następny blok do następnego górnika.

let registeredMiners = [];

let lastBlockMinedBy = null;

Zamierzasz również dodać dwa typy wiadomości.

• REQUEST_ALL_REGISTER_MINERS

• REGISTER_MINER

let MessageType = {

REQUEST_BLOCK: 'requestBlock',

RECEIVE_NEXT_BLOCK: 'receiveNextBlock',

RECEIVE_NEW_BLOCK: 'receiveNewBlock',

REQUEST_ALL_REGISTER_MINERS: 'requestAllRegisterMiners',

REGISTER_MINER: 'registerMiner'

};

Zanim zarejestrujesz swoich peerów jako górników, poprosisz o otrzymanie wszystkich istniejących zarejestrowanych górników w sieci, a następnie dodasz swojego partnera jako górnika w obiekcie RegisteredMiners. Robisz to, uruchamiając zegar, który aktualizuje swoich górników co pięć sekund.

setTimeout(function(){

writeMessageToPeers(MessageType.REQUEST_ALL_REGISTER_MINERS,

null);

}, 5000);

Teraz, gdy masz automatyczne polecenie limitu czasu, które może wskazywać na program obsługi, który aktualizuje listę zarejestrowanych górników, możesz również zautomatyzować polecenie, aby zarejestrować swojego partnera jako górnika;

setTimeout(function(){

registeredMiners.push(myPeerId.toString('hex'));

console.log('----------Register my miner --------------');

console.log(registeredMiners);

writeMessageToPeers(MessageType.REGISTER_MINER,

registeredMiners);

console.log('---------- Register my miner --------------');

}, 7000);

W swoim poleceniu switch chcesz zmodyfikować kod, aby móc ustawić programy obsługi dla przychodzących wiadomości dotyczących rejestracji górników. Chcesz śledzić zarejestrowanych górników, a także obsługiwać wiadomość po wydobyciu nowego bloku. Zobacz Listing dla obsługi górników.

Obsługa górników

case MessageType.REQUEST_ALL_REGISTER_MINERS:

```
console.log('-----------REQUEST_ALL_REGISTER_
MINERS------------- ' + message.to);
writeMessageToPeers(MessageType.REGISTER_MINER,
registeredMiners);
registeredMiners = JSON.parse(JSON.stringify(message.
data));
console.log('-----------REQUEST_ALL_REGISTER_
MINERS------------- ' + message.to);
break;
case MessageType.REGISTER_MINER:
console.log('-----------REGISTER_MINER------------- ' +
message.to);
let miners = JSON.stringify(message.data);
registeredMiners = JSON.parse(miners);
console.log(registeredMiners);
console.log('-----------REGISTER_MINER------------- ' +
message.to);
break;
```

Wyrejestruj górników

Musisz również wyrejestrować górnika, gdy połączenie z górnikiem zostanie zamknięte lub utracone.

```
console.log(`Connection ${seq} closed, peerId: ${peerId}`);
if (peers[peerId].seq === seq) {
delete peers[peerId];
console.log('--- registeredMiners before: ' + JSON.
stringify(registeredMiners));
let index = registeredMiners.indexOf(peerId);
if (index > -1)
registeredMiners.splice(index, 1);
console.log('--- registeredMiners end: ' + JSON.
stringify(registeredMiners));
}
```

```
});
```

Wykop nowy blok

W przeciwieństwie do bitcoina, który generuje blok co 10 minut, Twój blockchain zostanie ulepszony i będzie generował blok co 30 sekund. Aby to osiągnąć, zainstalowałeś już bibliotekę cron typu open source dla Node.js. Biblioteka cron działa tak samo jak cron w Linuksie. Możesz użyć biblioteki cron, aby ustawić częstotliwość ponownego wywoływania tego samego kodu, który będzie używany do wywoływania twoich górników co 30 sekund. Aby to zrobić, najpierw umieść bibliotekę w instrukcji import kodu na górze pliku p2p.js.

```
let CronJob = require('cron').CronJob;
```

Następnie możesz ustawić swoje cronjob tak, aby uruchamiało się co 30 sekund, a job.start(); rozpocznie pracę, jak pokazano na listingu

```
const job = new CronJob('30 ∗ ∗ ∗ ∗ ∗', function() {

let index = 0; // first block

if (lastBlockMinedBy) {

let newIndex = registeredMiners.indexOf(lastBlockMinedBy);

index = ( newIndex+1 > registeredMiners.length-1) ? 0 :

newIndex + 1;

}

lastBlockMinedBy = registeredMiners[index];

console.log('-- REQUESTING NEW BLOCK FROM: ' +

registeredMiners[index] + ', index: ' + index);

console.log(JSON.stringify(registeredMiners));

if (registeredMiners[index] === myPeerId.toString('hex')) {

console.log('-----------create next block -------------

----');

let newBlock = chain.generateNextBlock(null);

chain.addBlock(newBlock);

console.log(JSON.stringify(newBlock));

writeMessageToPeers(MessageType.RECEIVE_NEW_BLOCK,

newBlock);

console.log(JSON.stringify(chain.blockchain));

console.log('-----------create next block -------------

----');
```

```
}
});
job.start();
```

Przeglądając kod, zauważ, że indeks pierwszego bloku to 0, więc po wydobyciu pierwszego bloku zostanie ustawiony lastBlockMinedBy i będziesz prosić następnego górnika o kolejny blok.

```
let newIndex = registeredMiners.indexOf(lastBlockMinedBy);
index = ( newIndex+1 > registeredMiners.length-1) ? 0 : newIndex
+ 1;
```

Aby wygenerować i dodać nowy blok, wywołasz łańcuchy generateNextBlock i addBlock. Na koniec wyemitujesz nowy

```
let newBlock = chain.generateNextBlock(null);
chain.addBlock(newBlock);
writeMessageToPeers(MessageType.RECEIVE_NEW_BLOCK, newBlock);
```

W twoim kodzie twój przełącznik będzie obsługiwał nowe bloki przychodzące.

```
case MessageType.RECEIVE_NEW_BLOCK:
if ( message.to === myPeerId.toString('hex') && message.from
!== myPeerId.toString('hex')) {
console.log('-----------RECEIVE_NEW_BLOCK------------- '
+ message.to);
chain.addBlock(JSON.parse(JSON.stringify(message.data)));
console.log(JSON.stringify(chain.blockchain));
console.log('-----------RECEIVE_NEW_BLOCK------------- '
+ message.to);
}
break;
```

Aby zobaczyć ten kod w akcji, uruchom trzy wystąpienia swojego kodu.

```
> node p2p.js
```

Możesz zobaczyć komunikaty o zarejestrowaniu każdego peera jako górnika, a także twój kod, który zaczyna kopać bloki co 30 sekund w kolejności

Przechowywanie bloków w LevelDB

Jeśli uruchomisz swój blockchain na kilka godzin, zauważysz, że liczba tworzonych bloków rośnie, co może stać się problemem, ponieważ obecnie bloki te są przechowywane w pamięci podręcznej komputera. W miarę dodawania kolejnych bloków zużycie pamięci będzie rosło, a ostatecznie Twój

kod się zawiesi. Co więcej, bez przechowywania bloków w bazie danych nie będziesz w stanie uruchomić i zatrzymać sieci P2P, ponieważ bloki nie są zapisywane. Aby uwzględnić te i inne przypadki użycia, będziesz używać bazy danych LevelDB.

Uwaga : Baza danych LevelDB przechowuje pary nazwa-wartość w sposób zwany wyższy i niższy. Jest to idealna opcja dla sieci blockchain. W rzeczywistości bitcoin używa LevelDB do przechowywania nie tylko informacji o blokach, ale także informacji o transakcjach.

KROK 4: BAZA POZIOMU DO PRZECHOWYWANIA ĆWICZEŃ Z KLOCKAMI

LevelDB

W tym ćwiczeniu zaimplementujesz bazę danych do przechowywania bloków. Aby zacząć samodzielnie od poprzedniego kroku, będziesz używać wrappera Node.js LevelDB, dzięki czemu będziesz mógł komunikować się z LevelDB za pomocą swojego kodu. Zainstaluj bibliotekę przez npm.

```
> npm install level –save
```

Następnie utwórz katalog, w którym będziesz zapisywać bazę danych.

```
> mkdir db
```

Możesz teraz zaimplementować bazę danych. W swojej bibliotece chain.js dodasz kod, aby zapisać swój blok w bazie danych LevelDB, jak pokazano na Listingu

```
let level = require('level'),

fs = require('fs');

let db;

let createDb = (peerId) => {

let dir = __dirname + '/db/' + peerId;

if (!fs.existsSync(dir)){

fs.mkdirSync(dir);

db = level(dir);

storeBlock(getGenesisBlock());

}

}
```

Jak widać, używasz natywnej klasy __dirname Node.js, aby podać lokalizację ścieżki katalogu, ponieważ potrzebujesz pełnej ścieżki do zapisania bazy danych. Ponieważ uruchamiasz wiele instancji sieci P2P na tym samym komputerze, nie możesz użyć tej samej ścieżki dla każdego elementu równorzędnego, ponieważ baza danych musi być oddzielna. To, co możesz zrobić, to ustawić lokalizację każdej bazy danych w osobnej ścieżce, używając nazwy folderu jako nazwy twojego identyfikatora peera; następnie każda baza danych może być przechowywana w folderze db. Zauważ również, że zapisujesz pierwszy blok, getGenesisBlock(). Następnie tworzysz metodę storeBlock do przechowywania nowego bloku.

```
let storeBlock = (newBlock) => {

db.put(newBlock.index, JSON.stringify(newBlock), function (err) {
```

```
if (err) return console.log('Ooops!', err) // some kind
of I/O error
console.log('--- Inserting block index: ' + newBlock.
index);
})
}
```

Kiedy generujesz nowy blok za pomocą metody generateNextBlock, możesz teraz przechowywać blok w bazie danych LevelDB.

```
storeBlock(newBlock);
```

Zamierzasz również dodać metodę, aby móc pobrać blok z bazy danych LevelDB.

```
let getDbBlock = (index, res) => {
db.get(index, function (err, value) {
if (err) return res.send(JSON.stringify(err));
return(res.send(value));
});
}
```

Upewnij się, że udostępniasz metody createDb i getDbBlock.

```
if (typeof exports != 'undefined' ) {
exports.createDb = createDb;
exports.getDbBlock = getDbBlock;}
```

Wreszcie, w kodzie sieci P2P wystarczy utworzyć bazę danych po uruchomieniu kodu.

```
chain.createDb(myPeerId.toString('hex'));
```

Aby zobaczyć kod w akcji, uruchom instancję sieci P2P.

```
> node p2p.js
```

Możesz monitorować dane bazy danych w folderze db za pomocą polecenia tail z flagą -f. Terminal pozostanie otwarty i będzie mógł pokazywać nowe bloki podczas ich generowania.

```
> cd step4/db/[our peer Id]
> tail –f 000003.log
```

Wreszcie, w kodzie sieci P2P wystarczy utworzyć bazę danych po uruchomieniu kodu.

```
chain.createDb(myPeerId.toString('hex'));
```

Aby zobaczyć kod w akcji, uruchom instancję sieci P2P.

```
> node p2p.js
```

Możesz monitorować dane bazy danych w folderze db za pomocą polecenia tail z flagą -f. Terminal pozostanie otwarty i będzie mógł pokazywać nowe bloki podczas ich generowania . W tym ćwiczeniu utworzyłeś bazę danych LevelDB. Przechowujesz swoje bloki, więc będziesz mógł je odzyskać zamiast polegać na pamięci podręcznej. Utrzymuję rzeczy proste; gdyby to był naprawdę działający blockchain, zaimplementowałbyś następujące kroki:

1. Ogranicz wszystkie możliwe zagrożenia bezpieczeństwa.

2. Przechowuj i pobieraj swoje bloki z bazy danych LevelDB.

3. Utwórz metodę przywracania wpisów LevelDB.

4. Wyczyść stare bazy danych, ponieważ na każdym init tworzone są nowe.

Tworzenie portfela Blockchain

W kryptowalucie portfel jest niezbędny, aby nagradzać górników za generowanie bloków, a także aby móc tworzyć transakcje i wysyłać transakcje. W tej sekcji utworzysz portfel. Musisz utworzyć kombinację kluczy publicznych i prywatnych nie tylko w celu uwierzytelnienia użytkownika, ale także w celu przechowywania i pobierania danych, których jest właścicielem. Stworzysz portfel z kluczami publicznymi i prywatnymi. W przypadku bitcoin oryginalnym oprogramowaniem portfela jest podstawowy protokół bitcoin pobrany w rozdziale 2; potrzebuje całej księgi wszystkich transakcji od 2009 roku, która w chwili pisania ma ponad 150 GB. Z tego powodu większość używanych portfeli to portfele „lekkie" lub tzw. portfele z uproszczoną weryfikacją płatności (SPV), które synchronizują się z rdzeniem bitcoin. W blockchain dostępnych jest wiele różnych portfeli, od online po portfel papierowy, w którym zapisujesz swój klucz prywatny na kartce papieru. Zanim przejdziemy dalej, rzućmy okiem na to, jak możesz komunikować się z portfelem bitcoin. Jak pamiętasz, w Rozdziale 2 udało Ci się uzyskać saldo pewnego portfela bitcoin. Aby lepiej zrozumieć portfele, możesz utworzyć allet bitcoin za pomocą rdzenia bitcoin. Najpierw musisz uruchomić demona bitcoin.

> bitcoind – printtoconsole

Następnie możesz poprosić o adres.

> bitcoin-cli help getnewaddress

Następnie możesz zrzucić swoje klucze prywatne do pliku tekstowego.

> bitcoin-cli dumpwallet ~/mywallet.txt

Możesz uzyskać lokalizację swojego klucza prywatnego i wyświetlić klucz.

> vim /Users/[location]/mywallet.txt

Dla odniesienia, sprawdź kod podstawowego portfela bitcoin C++ tutaj:

> vim /[Bitcoin Core Location]/bitcoin/src/wallet/init.cpp

KROK 5: ĆWICZENIE Z PORTFELA

Utwórz portfel Blockchain

W tym ćwiczeniu wygenerujesz klucze publiczno-prywatne, które będą używane w Twoim portfelu. Będziesz używać implementacji biblioteki kryptografii krzywych eliptycznych do generowania kombinacji kluczy prywatnych i publicznych. Zauważ, że biblioteka krzywych eliptycznych używa secp256k1 jako algorytmu krzywej ECDSA.

};

Uwaga: Kryptografia krzywych eliptycznych (ECC) to technika szyfrowania kluczem publicznym stosowana przez bitcoin. Opiera się na teorii krzywych eliptycznych do generowania kluczy kryptograficznych. Secp256k1 to algorytm ECDSA z krzywą eliptyczną wykresu.

Aby zainstalować bibliotekę, uruchom następujące polecenie:

```
> npm install elliptic –save
```

Następnie dodaj plik i nazwij go wallet.js. Spójrz na pełny kod w aukcji

```
let EC = require('elliptic').ec,

fs = require('fs');

const ec = new EC('secp256k1'),

privateKeyLocation = __dirname + '/wallet/private_key';

exports.initWallet = () => {

let privateKey;

if (fs.existsSync(privateKeyLocation)) {

const buffer = fs.readFileSync(privateKeyLocation, 'utf8');

privateKey = buffer.toString();

} else {

privateKey = generatePrivateKey();

fs.writeFileSync(privateKeyLocation, privateKey);

}

const key = ec.keyFromPrivate(privateKey, 'hex');

const publicKey = key.getPublic().encode('hex');

return({'privateKeyLocation': privateKeyLocation,

'publicKey': publicKey});

};

const generatePrivateKey = () => {

const keyPair = ec.genKeyPair();

const privateKey = keyPair.getPrivate();

return privateKey.toString(16);

};
```

W pliku portfela tworzysz i inicjujesz kontekst EC.

```
const ec = new EC('secp256k1'),
```

Następnie przechowujesz lokalizację klucza prywatnego portfela, privateKeyLocation.

```
PrivateKeyLocation = __dirname + '/wallet/private_key';
```

Następnie możesz utworzyć metodę exports.initWallet do generowania rzeczywistego klucza publiczno-prywatnego, generatePrivateKey.

```
const keyPair = ec.genKeyPair();
```

```
const privateKey = keyPair.getPrivate();
```

Zauważ, że będziesz generować nowy portfel tylko wtedy, gdy taki nie istnieje.

```
if (fs.existsSync(privateKeyLocation))
```

W tym ćwiczeniu utworzysz plik wallet.js, wykorzystując bibliotekę kryptografii krzywych eliptycznych do wygenerowania kombinacji kluczy prywatny-publiczny.

Aby zobaczyć, jak działa kod, tymczasowo dodaj następujący kod na końcu pliku wallet.js. Skrypt utworzy klucze publiczne i prywatne.

```
let wallet = this;

let retVal = wallet.initWallet();

console.log(JSON.stringify(retVal));
```

Następnie utwórz katalog portfela do przechowywania klucza prywatnego i uruchom skrypt. Kod zainicjuje skrypt i utworzy twój klucz publiczny.

```
> mkdir wallet

> node wallet.js

> cat wallet/private_key
```

Pamiętaj, aby zakomentować te wiersze, ponieważ w następnym ćwiczeniu utworzysz interfejs API, aby móc tworzyć klucze za pomocą przeglądarki.

Tworzenie API

Następnym krokiem jest utworzenie interfejsu programu aplikacji (API), aby móc uzyskać dostęp do kodu, który piszesz. Jest to ważna część łańcucha bloków, ponieważ chcesz uzyskać dostęp do swoich bloków i portfela lub dowolnej innej operacji sieciowej P2P za pomocą usługi HTTP. W tej sekcji będziesz korzystać z biblioteki ekspresowej, ponieważ jest ona łatwa w obsłudze i będziesz mógł łatwo tworzyć swoje API.

KROK 6: ĆWICZENIE Z ŁAŃCUCHEM BLOKÓW API P2P

Tworzenie API

W tym ćwiczeniu utworzysz interfejs API do interakcji z siecią blockchain P2P.

Będziesz tworzyć następujące usługi:

- blocks: pobiera wszystkie bloki w łańcuchu bloków
- getBlock: pobiera określony blok według indeksu
- getDBBlock: Pobiera blok z bazy danych

• getWallet: Tworzy nowy portfel poprzez wygenerowanie klucza publiczno-prywatnego

Zainstalujesz express i body-parser. Te biblioteki pozwolą Ci stworzyć serwer i wyświetlać strony w przeglądarce.

> npm install express body-parser --save

Musisz również zaimportować utworzony plik wallet.js.

```
let express = require("express"),

bodyParser = require('body-parser'),

wallet = require('./wallet');
```

Następnie tworzysz metodę o nazwie initHttpServer, która zainicjuje serwer i tworzenie usług. Ponieważ korzystasz z różnych instancji sieci P2P i uruchamiasz instancje na tym samym komputerze, chcesz używać różnych numerów portów. Często używany jest port 80 lub 8081 dla usług HTTP, ale nie jest to wymagane. To, co zrobisz, to przekażesz losowy numer portu, którego używasz, i użyj metody plasterka, aby uzyskać dwie ostatnie cyfry numeru portu.

```
let initHttpServer = (port) => {

let http_port = '80' + port.toString().slice(-2);

let app = express();

app.use(bodyParser.json());
```

Usługa Bloki pobierze wszystkie Twoje bloki.

```
app.get('/blocks', (req, res) => res.send(JSON.stringify(

chain.blockchain )));
```

Usługa getBlock będzie pobierać jeden blok na podstawie indeksu.

```
app.get('/getBlock', (req, res) => {

let blockIndex = req.query.index;

res.send(chain.blockchain[blockIndex]);

});
```

Usługa getDBBlock będzie pobierać wpis bazy danych LevelDB na podstawie indeksu.

```
app.get('/getDBBlock', (req, res) => {

let blockIndex = req.query.index;

chain.getDbBlock(blockIndex, res);

});
```

Usługa getWallet będzie wykorzystywać plik wallet.js utworzony w poprzednim kroku i wygeneruje parę kluczy publiczny-prywatny.

```
app.get('/getWallet', (req, res) => {
```

```
res.send(wallet.initWallet());

});
```

Na koniec skorzystasz z metody słuchania ekspresowego.

```
app.listen(http_port, () => console.log('Listening http on

port: ' + http_port));

};
```

Wywołasz metodę initHttpServer, którą utworzyłeś po uruchomieniu sieci P2P i wybraniu losowego portu.

```
(async () => {

const port = await getPort();

initHttpServer(port);

}
```

Aby wywołać swoje usługi, uruchom sieć P2P, a następnie otwórz przeglądarkę i wywołaj API.

http://localhost:80[port]/getWallet

http://localhost:80[port]/blocks

http://localhost:80[port]/getBlock?index=0

http://localhost:80[port]/ getDBBlock?index=0

W tym ćwiczeniu utworzyłeś usługi API i możesz teraz wchodzić w interakcję z siecią P2P. Stworzyłeś swoje usługi, dzięki czemu będziesz mógł tworzyć wiele instancji sieci P2P na tej samej maszynie; jednak w rzeczywistości każda maszyna będzie obsługiwać tylko jednego peera. W następnym ćwiczeniu utworzysz interfejs wiersza poleceń (CLI), aby łatwo wywoływać te usługi.

Tworzenie interfejsu wiersza poleceń

W ostatnim kroku tego rozdziału będziesz tworzył interfejs wiersza poleceń (CLI). CLI jest potrzebny, aby móc łatwo uzyskać dostęp do utworzonych usług. Nie będę wchodził w cały wewnętrzny proces skryptu CLI, ponieważ wykracza to poza zakres tego rozdziału; jednak możesz pobrać cały przykład i przejrzeć go.

KROK 7: ĆWICZENIE CLI

Polecenie blokowe

W tym ćwiczeniu utworzysz CLI do interakcji i dostępu do sieci blockchain P2P. Następnie zainstaluj biblioteki, których będziesz używać do uruchamiania obietnic, uruchamiania funkcji asynchronicznej, dodawania kolorów do konsoli i przechowywania plików cookie.

```
> npm babel-polyfill async update-notifier handlebars colors

nopt –save
```

W poleceniu block.js ustawisz dwie komendy: get i all. Spójrz na cały kod na listingu

```
let logger = require('../logger');
function Block(options) {
this.options = options;
}
Block.DETAILS = {
alias: 'b',
description: 'block',
commands: ['get', 'all'],
options: {
create: Boolean
},
shorthands: {
s: ['--get'],
a: ['--all']
},
payload: function(payload, options) {
options.start = true;
},
};
Block.prototype.run = function() {
let instance = this,
options = instance.options;
if (options.get) {
instance.runCmd('curl http://localhost:' + options.argv.
original[2] + '/getBlock?index=' + options.argv.original[3]);
}
if (options.all) {
instance.runCmd('curl http://localhost:' + options.argv.
original[2] + '/blocks');
}
};
```

```
Block.prototype.runCmd = function(cmd) {
const { exec } = require('child_process');
logger.log(cmd);
exec(cmd, (err, stdout, stderr) => {
if (err) {
logger.log(`err: ${err}`);
return;
}
logger.log(`stdout: ${stdout}`);
});
};
exports.Impl = Block;
```

Jak widać, polecenie wallet.js będzie zawierać get i wszystkie metody wskazujące na polecenie curl w celu uruchomienia wywołania usługi HTTP.

Polecenie portfela

Podobnie polecenie block.js będzie zawierało metodę create i polecenie curl do uruchomienia wywołania usługi HTTP. Zobacz listing

```
let logger = require('../logger');
function Wallet(options) {
this.options = options;
}
Wallet.DETAILS = {
alias: 'w',
description: 'wallet',
commands: ['create'],
options: {
create: Boolean
},
shorthands: {
c: ['--create']
},
payload: function(payload, options) {
```

```
options.start = true;

},

};

Wallet.prototype.run = function() {

let instance = this,

options = instance.options;

if (options.create) {

instance.runCmd('curl http://localhost:' + options.argv.

original[2] + '/getWallet');

}

};

Wallet.prototype.runCmd = function(cmd) {

const { exec } = require('child_process');

logger.log(cmd);

exec(cmd, (err, stdout, stderr) => {

if (err) {

logger.log(`err: ${err}`);

return;

}

logger.log(`stdout: ${stdout}`);

});

};

exports.Impl = Wallet;
```

Teraz, gdy masz już skonfigurowane polecenia, możesz dodać swój CLI do bash_profile jako alias, aby móc uruchomić CLI z dowolnej lokalizacji ścieżki.

```
> vim ~/.bash_profile

alias cli='node /[project location]/step7/bin/bin/cli.js
```

Zapisz i uruchom bash_profile, aby zastosować te zmiany.

```
> . ~/.bash_profil
```

Możesz zadzwonić do CLI po uruchomieniu P2P i poznaniu używanych portów.

```
> cli block --get [port] 1 #port #index
```

> cli block –all [port] #port

> cli wallet --create [port]

Na przykład uruchom instancję sieci P2P w Terminalu.

> node p2p.js

Następnie otwórz nowy terminal okna i uruchom polecenie CLI, aby pobrać pierwszy wygenerowany blok.

> blok cli --get [port] 1

W tym ćwiczeniu utworzyłeś dwa polecenia do pobierania bloków i tworzenia portfela. Jest to punkt wyjścia dla twojego CLI i będziesz mógł kontynuować dodawanie poleceń w razie potrzeby.

Dokąd się udać?

Wspomniałem już, że kod w tym rozdziale nie bierze pod uwagę wielu przypadków użycia i nie ma żadnych zabezpieczeń, aby był prosty. Jest wiele rzeczy, które możesz zrobić, aby ulepszyć kod.

• Potwierdzenia: Każdy górnik wysyła wiadomość z blokiem. Możesz stworzyć system potwierdzeń, aby zapewnić integralność danych.

• Transakcje/dane: Możesz zaimplementować transakcje lub obiekty danych, aby rozwiązać problem podwójnych wydatków, walidacji transakcji i transakcji coinbase.

• levelDB: Po zainicjowaniu P2P można utworzyć skrypt do pobierania i zapisywania wszystkich bloków w bazie danych LevelDB, sprawdzania ich poprawności i czyszczenia bazy danych w razie potrzeby.

Streszczenie

W tej części omówiono, jak stworzyć własną podstawową sieć blockchain P2P; byłeś w stanie wysyłać i odbierać wiadomości oraz umieszczać bloki w tych wiadomościach. Udało Ci się zarejestrować i wyrejestrować górników oraz wdrożyć prosty mechanizm konsensusu PoS. Utworzyłeś nowe bloki i wysłałeś je między rówieśnikami. Można również skonfigurować bazę danych typu nazwa-wartość LevelDB do przechowywania bloków. Kontynuowałeś i utworzyłeś portfel składający się z par kluczy prywatny-publiczny. Wreszcie stworzyłeś sposoby komunikowania się z siecią P2P za pośrednictwem usług API i CLI. W następnym rozdziale zagłębisz się w zrozumienie portfeli i transakcji bitcoin poprzez interakcję z podstawowym API bitcoin.