4 Block information when sending IPDC （2-b)

As part of the research on block headers and Merkle paths on the Cardano platform, we conducted investigations into block validation, rollback, and transaction root analysis.

4.1 Research on Block Confirmation

4.1.1 Research Objective

Monitoring the Integrity of the Cardano Blockchain in Real-Time and Investigating Inconsistency Detection Processes
We investigate methods for ensuring the security and reliability of the blockchain, particularly through the practice of retrieving block information and detecting inconsistencies using the Blockfrost API.

<Practical Real-Time Monitoring of Block Information>
Since blockchain operations change in real-time, periodic monitoring is crucial. In this research, we explore a method for real-time monitoring, where the latest block information is retrieved every 15 seconds, and the continuity of blocks and the integrity of block hashes are checked. This technology provides a foundation for always keeping track of the blockchain's operational status and responding promptly.

<Validation of Inconsistency Detection Mechanisms>
To maintain the integrity of the blockchain, it is essential to quickly detect inconsistencies when they occur. This research validates mechanisms for detecting inconsistencies when block hashes do not match or when gaps occur in block heights. By doing so, we verify practical methods for ensuring the security and reliability of the blockchain.

4.1.2 Libraries used and environment

<Libraries used>

fetch API

: It is used to make HTTP requests and retrieve data from external APIs using the native JavaScript API.

<environment>

It works in both browser and Node.js environments.
To use the Blockfrost API, you need to obtain an API key in advance.

4.1.3 Basic Research Content

Initial Setup

```
const apiKey = process.env.PROJECT_ID; // BlockfrostのAPIキー
let lastHeight = 0;
let lastHash = "";
```

API KEY

：Define the key to access the Blockfrost API. A key for the preprod network is specified.

State Management Variables

：lastHeight and lastHash hold the height and hash of the previously retrieved block.

Obtaining and Monitoring Block Information

```
const fetchBlockInfo = async () => {
    try {
        const response = await fetch('https://cardano-preprod.blockfrost.io/api/v0/blocks/latest', {
            method: 'GET',
            headers: {
                'project_id': apiKey
            }
        });

        if (response.ok) {
            const blockInfo = await response.json();
            const currentHeight = blockInfo.height;

            if (lastHeight !== currentHeight) {

                if (lastHeight > 0 && currentHeight > lastHeight + 1) {
                    for (let h = lastHeight + 1; h < currentHeight; h++) {
                        await fetchBlockDetails(h);
                    }
                }
                if (lastHash !== blockInfo.previous_block) {
                    console.log("■■■■ DIFFERENT HASH ■■■■");
```

```
                }
                console.log("prevHash:", blockInfo.previous_block);
                console.log('Latest Block information:', blockInfo);
                console.log("height:", blockInfo.height);
                console.log("calcHash:", blockInfo.hash);
                lastHash = blockInfo.hash;
                lastHeight = currentHeight;
            } else {
                console.log("skip:", blockInfo.height);
            }
        } else {
            const error = await response.text();
            console.error('Error fetching block information:', error);
        }
    } catch (error) {
        console.error('Error:', error);
    }
};
```

fetchBlockInfo Function

: Retrieves the latest block information and compares it with the previously retrieved block.

Inconsistency Check

: Logs any inconsistencies if the block height or the previous block's hash does not match.

Gap Check

: If there is a gap between the previously retrieved block height and the latest block height, the blocks in between are retrieved individually to check the details.

Getting Blocks with Gaps

```
const fetchBlockDetails = async (height) => {
    try {
        const response = await fetch(`https://cardano-preprod.blockfrost.io/api/v0/blocks/${height}`,
{
            method: 'GET',
            headers: {
                'project_id': apiKey
            }
        });

        if (response.ok) {
            const blockDetails = await response.json();
            if (lastHash !== blockDetails.previous_block) {
                console.log("■■■■ DIFFERENT HASH ■■■■");
            }
            console.log("prevHash:", blockDetails.previous_block);
            console.log('Missing Block information:', blockDetails);
            console.log("height:", blockDetails.height);
            console.log("calcHash:", blockDetails.hash);
            lastHash = blockDetails.hash;
        } else {
            const error = await response.text();
            console.error(`Error fetching block information for height ${height}:`, error);
        }
    } catch (error) {
        console.error(`Error fetching block information for height ${height}:`, error);
```

fetchBlockDetails 関数

：Retrieve block information corresponding to the specified block height and compare it with the previous hash.

Periodic Block Information Getting

```
setInterval(fetchBlockInfo, 15000); // 15000ミリ秒（15秒）ごとに実行
fetchBlockInfo();
```

setInterval

：Execute the fetchBlockInfo function every 15 seconds to periodically monitor the latest block information.

4.1.4 Research Results

This code retrieves the latest block information on the Cardano preprod network and monitors block continuity and consistency. Every 15 seconds, the latest block information is retrieved, and the following details are checked:

- **Block Height**: Compare with the height of the previously retrieved block to check for gaps.
- **Block Hash**: Compare with the previous block hash to check for inconsistencies.

4.2 Research on Rollback

4.2.1 Research Objective

Verifying a Practical Process for Maintaining the Integrity of the Cardano Blockchain and Building a Reliable Monitoring System.
By introducing hash verification and reconciliation functions using past data, this enables swift troubleshooting in the event of inconsistencies.

<Advanced Integrity Maintenance of the Cardano Blockchain>
The integrity of the Cardano blockchain is ensured by the continuity of blocks and the consistency of hashes. The purpose of this research is to build advanced monitoring functions that detect and address inconsistencies, such as hash mismatches or missing blocks on the blockchain. In particular, by introducing the storage of past block hashes and reconciliation (integrity recovery) functions, the continuity of the blockchain is maintained, further strengthening its reliability.

4.2.2 Libraries used and environment

There are no libraries to be used.

4.2.3 Basic Research Content

Functionality Overview

The following new features have been added:

- **Hash History Storage**: Save the hashes of previously retrieved blocks and use them during integrity checks.
- **Block Reconciliation**: A function that, when a hash mismatch is detected on the blockchain, goes back through previous blocks to find the matching block.

Hash History Storage

```
let pastHashes = [];
```

```
pastHashes.push({ height: blockInfo.height, hash: blockInfo.hash });
if (pastHashes.length > 100) {
    pastHashes.shift();
}
```

Managing Past Hashes

: Introduce a pastHashes list to store the hashes and heights of previously retrieved blocks. This list holds up to 100 hash entries, and older entries are deleted when this limit is exceeded. This provides a foundation for comparing the hashes of past blocks with the current block.

Adding Hashes and Managing the List

: Add the information of newly retrieved blocks to pastHashes. If the list exceeds 100 entries, the oldest entry is removed to manage memory usage.

Reconciliation in Case of Hash Mismatch

```
const reconcileBlocks = async (startHeight) => {
    console.log("Reconciling blocks...");
    let currentHeight = startHeight;

    while (currentHeight > 0) {
        const blockDetails = await fetchBlock(`https://cardano-
preprod.blockfrost.io/api/v0/blocks/${currentHeight}`);
        if (!blockDetails) break;

        const pastBlock = pastHashes.find(p => p.height === blockDetails.height);

        if (pastBlock && pastBlock.hash === blockDetails.hash) {
            console.log(`Match found at height ${currentHeight}`);
            lastHeight = currentHeight;
            lastHash = blockDetails.hash;
            break;
        } else {
            console.log(`No match found at height ${currentHeight}, fetching previous block`);
            currentHeight--;
        }
    }
};
```

Reconciliation Function:

: This function is called when a hash mismatch is detected. It retrieves block information by going back from the current block height and compares it with the data stored in pastHashes. If a block with a matching hash is found, that block is used as the basis for restoring consistency.

Utilizing Past Block Information:

: By comparing with the past block information stored in the pastHashes list, block integrity can be efficiently verified, and the point of inconsistency can be identified.

Improved Block Monitoring Functionality

```
const fetchBlockInfo = async () => {
    const blockInfo = await fetchBlock('https://cardano-preprod.blockfrost.io/api/v0/blocks/latest');
    if (!blockInfo) return;

    const currentHeight = blockInfo.height;

    if (lastHeight !== currentHeight) {
        if (lastHeight > 0 && currentHeight > lastHeight + 1) {
```

```
            for (let h = lastHeight + 1; h < currentHeight; h++) {
                await fetchBlockDetails(h);
            }
        }

        if (lastHash !== blockInfo.previous_block) {
            console.log("■■■■ DIFFERENT HASH ■■■■");
            await reconcileBlocks(lastHeight);
        }

        updateState(blockInfo);
    } else {
        console.log("skip:", blockInfo.height);
    }
};
```

Enhanced Inconsistency Handling:

: When block hashes do not match, instead of just displaying a warning, the reconcileBlocks function is called to attempt to restore consistency. This process ensures the continuity of the blockchain and improves the system's reliability.

Benefit from improved code

Improved Reliability:

: When block inconsistencies occur, referencing past block data helps restore appropriate consistency, thereby enhancing the reliability of the blockchain.

Efficient Monitoring:
: By storing past hashes and using them in the reconciliation process, troubleshooting becomes easier when inconsistencies are detected.

4.2.4 Research Results

With this code, the Cardano blockchain monitoring system has gained advanced functionality to maintain block hash integrity and address potential inconsistencies. The addition of storing past block hashes and the reconciliation function further enhances the reliability of the blockchain and strengthens real-time monitoring.

4.3 Research on Transaction Route

4.3.1 Research Objective

The purpose of this research is to understand and apply the construction of a Merkle Tree and the use of the Blake2b hash function to verify the integrity of transaction data on the Cardano blockchain.

<Practical Application of Efficient Data Verification Using Merkle Tree>

The Merkle Tree is widely used as a method for efficiently verifying large datasets. In this research, we will confirm how to construct a Merkle Tree using transaction IDs on the Cardano blockchain, hashed with the Blake2b hash function.

<Application and Understanding of the Blake2b Hash Function>

In blockchain technology, hashing data is essential for ensuring security and integrity. The goal of this research is to understand how to hash transaction IDs using the Blake2b hash function and apply the results to construct a Merkle Tree. Blake2b is known for being a fast and secure hash function, and by applying it in practice, we will confirm its advantages in data processing on the blockchain.

4.3.2 Libraries used and environment

<Libraries used>

axios

: Used to make HTTP requests and send transactions to the Blockfrost API.

dotenv

: Used to load environment variables.。

<environment>

This is JavaScript code that runs in the Node.js environment.
An API key is required to use the Blockfrost API.
The blakejs library is required to use the Blake2b hash function.

4.3.3 Basic Research Content

Getting Transaction

```
async function fetchBlockTransactions(height, apiKey) {
    try {
        const response = await axios.get(`https://cardano-
preprod.blockfrost.io/api/v0/blocks/${height}/txs`, {
            headers: { 'project_id': apiKey }
        });
        return response.data;
    } catch (error) {
        console.error('Error fetching block transactions:', error);
        return [];
    }
}
```

fetchBlockTransactions Function

: Retrieves transaction IDs using the Blockfrost API based on the specified block height. Authentication is performed using the API key (apiKey).

Error Handling

: If the API request fails, an error message is displayed and an empty array is returned.

トランザクション ID のハッシュ計算

```
function getTransactionHashes(transactions) {
    return transactions.map(tx => blake.blake2bHex(tx, null, 32));
}
```

getTransactionHashes Function

: Hashes the retrieved transaction IDs using the Blake2b hash function, generating 32-byte hash values. This sets the foundation for constructing a Merkle Tree from the transaction IDs.

Hash Concatenation and Merkle Tree Construction

```
function hashConcat(left, right) {
    return blake.blake2bHex(left + right, null, 32);
}

function buildMerkleTree(hashes) {
    let tree = [hashes];

    while (tree[tree.length - 1].length > 1) {
        let currentLevel = tree[tree.length - 1];
        let nextLevel = [];

        for (let i = 0; i < currentLevel.length; i += 2) {
            const left = currentLevel[i];
            const right = i + 1 < currentLevel.length ? currentLevel[i + 1] : left;
            nextLevel.push(hashConcat(left, right));
        }
        tree.push(nextLevel);
    }
    return tree;
}
```

hashConcat Function

: Concatenates the left and right hashes, then re-hashes them using the Blake2b hash function. This is used when constructing the upper levels of the Merkle Tree.

buildMerkleTree Function

: Constructs a Merkle Tree from the given transaction hashes. At each level, from the leaves to the root, it concatenates the hashes, ultimately generating a single root hash (Merkle Root).

Getting Transactions and Constructing a Merkle Tree

```
(async () => {
    const apiKey = process.env.PROJECT_ID; // BlockfrostのAPIキー
    const height = 2528206;

    const transactions = await fetchBlockTransactions(height, apiKey);
    if (transactions.length === 0) {
        console.error('No transactions found');
        return;
    }
```

```
    console.log('Transaction IDs:', transactions);
    const transactionHashes = getTransactionHashes(transactions);
    const tree = buildMerkleTree(transactionHashes);
    console.log('Merkle Tree:');
    tree.forEach((level, index) => {
        console.log(`Level ${index}:`, level);
    });

    const root = tree[tree.length - 1][0];
    console.log('Merkle Root:', root);
})();
```

Transaction Retrieval

: Call the fetchBlockTransactions function to retrieve the transactions for the specified block.

Hashing Transaction IDs

: Hash the retrieved transaction IDs using the getTransactionHashes function.

Constructing the Merkle Tree

: Use the buildMerkleTree function to construct a Merkle Tree based on the hashed transaction IDs.

Displaying the Merkle Root

: Display the root hash (Merkle Root) of the constructed Merkle Tree.

4.3.4 Research Results

This code retrieves transactions from the specified block, hashes the transaction IDs using the Blake2b hash function, and constructs a Merkle Tree. Finally, it calculates the Merkle Root and outputs the result.

Transaction Retrieval

: Transaction IDs are retrieved from the specified block and used for hashing.

Merkle Tree Construction

: Each transaction ID is hashed, and these are used to progressively construct the Merkle Tree.

Merkle Root Calculation

: The hashes at the top level are concatenated, and the final Merkle Root is obtained.