Spartan Gold's Additional Features

Vincent Nguyen, Anh Dinh

San Jose State University

Department of Computer Science

CS 168: Blockchain and Cryptocurrency

Prof. Austin Thomas

Spring 2023

Overview

In this project we will be implementing some features available to Bitcoin that aren't immediately available to the cryptocurrency Spartan Gold. The purpose of this project is to enhance the functionality of Spartan Gold and make it a bit more robust in the capabilities of the cryptocurrency.

First, we will need to take the time to completely understand the features that we plan to implement that are currently in Bitcoin. We will be taking a look at using Merkle Trees to store transactions, looking at fixed block size, and variable Proof-of-work. After analyzing these functions and features in Bitcoin we will begin our own adaptation of these features in Spartan Gold.

Once we finish the implementation of these features we will take the time to test that the functions work with Spartan Gold. We will use test methods similar to those practiced in class like when using a fake network, sample clients and miners, and seeing how the functions work with these different variables.

Objectives and Scope

The objectives of this project are:

1. Implement Merkle Trees to Store Transactions
    a. This will hopefully provide a more efficient and secure method of verifying transaction integrity
2. Implement a Fixed Block Size

    a. This will potentially provide more predictable transaction processing times. Although Bitcoin regularly adjusts its time to take 10 minutes, this may help to reduce the risks of network congestion.

3. Implement a Variable Proof-of-Work
    a. We are hoping that this implementation will help to adjust the difficulty of mining based on the network demands.
    b. Ensure that the network can adapt to changes when it comes to transaction volume and maintain security
4. Test
    a. Finally, we will test these features to ensure that they all work together with Spartan Gold

The scope of this project will be limited to the implementation of these features and objectives. This project will involve researching the necessary information, developing the project, and testing all the features. We won't be having any major changes to the overall infrastructure of Spartan Gold or Bitcoin. We will simply be adding more capabilities to Spartan Gold.

Implementation

Variable Proof-of-Work

Implementing variable proof-of-work is important because we don't want a powerful network to be able to continuously mine blocks at a faster and faster rate. Currently, Spartan Gold has a static target that miners will need to hit to successfully prove that they have mined a

block. This means that with a powerful machine, blocks can be mined out rather quickly, this is something we want to change. Currently, Bitcoin has a PoW system that increases or decreases the difficulty of the PoW target to keep the time to produce a block to around ten minutes. So, every ten minutes a miner somewhere is able to produce a block and earn new bitcoin. However, in Spartan Gold, blocks are currently mined out nearly every half a second. So for our purposes we will try to keep the time to discover blocks to around ten seconds. These are the steps:

- First we will do this by creating a constant called DESIRED_BLOCK_TIME and setting it to 10,000 ms which is equivalent to 10 seconds.

```
// 10 seconds
const DESIRED_BLOCK_TIME = 10000;
```

- Secondly we will create a method called “updateTarget” which will take in the prevBlock as an argument. It will check if the prevBlock is there and continue to get the time difference between the previous block and the current block being created. Once the time difference is calculated it will be measured against the DESIRED_BLOCK_TIME of 10 seconds. If the amount of time is less than 10 seconds, then the difficulty will be increased. If the amount of time is more than 10 seconds then the difficulty will be decreased. The difficulty of the target will be changed by converting the target into a Number, which will then be rounded down by using Math.floor, then turning it into a BigInt which matches the target. It will then either be increased or decreased by 50% to see a change in the amount of time it takes to mine a block.

```
/**
 * This will change the difficulty of the Proof-of-Work target.
 * Since we want the block time to take around 10 seconds, whenever
 * the time difference between the current block and the previous block
 * are more or less than 10 seconds, the target will be decreased or
 * increased respectively.
 *
 * @param {*} prevBlock
 * @returns
 */
updateTarget(prevBlock) {
  // Check to see if previous block is there
  if (!prevBlock) return;

  // Time difference between the current block and the previous block
  const timeDiff = this.timestamp - prevBlock.timestamp;

  // If the time difference is less than 10 seconds, increase the difficulty by 50%
  if (timeDiff < DESIRED_BLOCK_TIME) {

    // Convert the target into a "Number", round down using Math.floor, create a BigInt to turn into target
    this.target = BigInt(Math.floor(Number(this.target) * 0.5));

  // if the time difference is greater than 10 seconds, decrease the difficulty by 50%
  } else if (timeDiff > DESIRED_BLOCK_TIME) {

    // Convert the target into a "Number", round down using Math.floor, create a BigInt to turn into target
    this.target = BigInt(Math.floor(Number(this.target) * 1.5));
  }
}
```

- Thirdly, the updateTarget method will be called in the constructor of the block. So whenever a block is being created, the target will be updated based on the time difference, eventually leading to a 10 second block time.

```
// Added a call to update the target
this.updateTarget(prevBlock);
```

Merkle Trees for Storage

Merkle trees are an integral part of a cryptocurrency because they provide many levels of security and integrity for the blockchain that is desired. Currently, Spartan Gold stores the blockchain on seemingly a linked list or map data structure. This gives an opportunity to

implement Merkle trees to improve on data integrity, security, efficient ways of verification, and making it harder for people to tamper with the data. These are the steps to do it:

- First, we will create a simple implementation of a Merkle Tree. We will create a new class called MerkleTree, in which the constructor takes in a list of transactions to build the tree. It starts by initializing an array called tree and calling buildTree(transactions) to build the tree.

```
module.exports = class MerkleTree {
  constructor(transactions) {
    this.tree = [];
    this.buildTree(transactions);
  }
```

- Second, we will create a helper method called hash which will hash the data provided to it, in this case it will be the id of the transactions.

```
// Hashes the id of the transaction
hash(data) {
  return crypto.createHash('sha256').update(data).digest('hex');
}
```

- Third, we will implement the buildTree method which takes in a list of transactions to build the tree. It will begin by using the transactions, getting their ids and hashing them, mapping them to each appropriate transaction. Next, we will push the hashes onto the tree array initialized in the constructor. Next, we will build the Merkle Tree by combining two hashes at a time then hashing them together to form the next level. If there are an even amount of hashes, then each pair will be concatenated and hashed together to form the hash in the next level. If there are an odd number of hashes, then the last lonely hash will

be hashed with itself. All of these new hashes are stored in a temporary array, and pushed to the tree array in the end.

```
// Builds the tree by hashing the id of each transaction and pushing it onto the tree.
// Then repeatedly combining each pair of hashs by concatenating them and hashing the result
// until only one hash is left. if odd number of hashes, the last one will hash with itself
buildTree(transactions) {

  // Create an array of hashes by using the keys of transaction and h
  let hashes = Array.from(transactions.keys()).map(id => this.hash(id));
  this.tree.push(hashes);

  // While there are hashes in the hash array, combine the hashes to build the merkle tree
  while (hashes.length > 1) {

      // Store the new hashes in here for each of the levels
      let temp = [];

      // Iterate over the hashes array by 2s to combine hashes
      for (let i = 0; i < hashes.length; i += 2) {

          // if there are an even amount of hashes it will concatenate them and hash them together
          if (i + 1 < hashes.length) {
              temp.push(this.hash(hashes[i] + hashes[i + 1]));

          // else if there are an odd number of hashes the last one will hash with itself
          } else {
              temp.push(this.hash(hashes[i] + hashes[i]));
          }
      }
    // stores each level of the hashes in the tree array
    hashes = temp;
    this.tree.push(hashes);
  }
}
```

- Lastly, the display method which will display the Merkle Tree after it is built.

```
  // Displays the merkle tree in a JSON format, showing each of the levels
  // Level 0 being the leaves, then climbing each level until the root
  display() {

    // create a new treeData array that creates the tree, i being the level of the tree,
    // and level being the hashes contained on each level
    const treeData = this.tree.map((level, i) => ({
        level: i,
        values: level
    }));

    // Converts the treeData into a JSON string
    const formattedTreeData = JSON.stringify(treeData, null, 2);

    // print to console
    console.log(formattedTreeData);
  }

  // Get the root hash of the Merkle tree
  getRootHash() {
    return this.tree[this.tree.length - 1][0];
  }
}
```

Fixed Block Size

Fixed block size is important to Bitcoin because it helps to maintain a predictable and consistent block creation time, it helps with resource management, and helps with maintaining the decentralized ability of Bitcoin. Currently, Spartan Gold doesn't have a fixed block size, so each block can be a varying amount of size depending on each size of the transaction. The current limit of Bitcoin's block size is limited to 1 MB, however, this can be roughly 2000 transactions, which is very large for our purposes. So we will play with numbers between 0-10 transactions per block. This is how we did it:

- First, we set a constant MAX_TRANSACTIONS_PER_BLOCK and we are able to set this to whatever limit we want it to be. For this example we used a limit of 8.

```
const MAX_TRANSACTIONS_PER_BLOCK = 8;
```

- Second, we put a constraint in addTransaction of the block class to check whether the transactions are equal to the MAX_TRANSACTIONS_PER_BLOCK we set. If so, then we return false and stop any more transactions being added to the block.

```
addTransaction(tx, client) {
  // Check if the block has reached the maximum number of transactions
  if (this.transactions.size >= MAX_TRANSACTIONS_PER_BLOCK) {
    if (client) client.log(`Maximum number of ${MAX_TRANSACTIONS_PER_BLOCK} transactions reached for block ${this.id}.`);
    return false;
```

Difficulties

Implementing different Bitcoin features with Spartan Gold can be a challenging, but rewarding endeavor. The first challenge we had to overcome was implementing a variable Proof-of-Work. This was initially pretty difficult because we had to figure out how to measure the difference in time between blocks. Navigating the different fields that a block had, figuring out how the target was initially set, how to manipulate the target to make it easier or harder, and how to implement it.

The next major challenge that was presented to us was implementing the Merkle Trees. We had a previous lab on Merkle Trees, so the understanding of how the Merkle Tree worked was already there, so we initially thought this wouldn't be too much trouble. However, the Merkle Trees became the most challenging aspect of this project. Using the previous lab Merkle Tree and integrating it into Spartan Gold became more difficult than simply starting from scratch and creating a separate implementation of the Merkle Tree. However, in time we were able to write a MerkleTree class that could build the tree using a list of transactions and display it.

Setting the fixed block size was actually the least challenging because we used transactions as a way to limit the size of each block. However, trying to limit the size of the block based on the actual amount of memory size like Bitcoin's 1 MB block size, would have been much more challenging.

Future Work

Implementing these features into Spartan Gold was a challenging, but rewarding experience. In the future we would love to implement more features or optimize the current features that we implemented. Our implementation of the Merkle Tree is pretty simple, in which it only builds and displays the Merkle Tree. It doesn't have verification processes, which is a major portion of Merkle Trees.

References

Bitcoin. (n.d.). *Bitcoin/Bitcoin: Bitcoin Core Integration/Staging tree*. GitHub. https://github.com/bitcoin/bitcoin

GeeksforGeeks. (2022, September 9). *Introduction to Merkle Tree*. GeeksforGeeks. https://www.geeksforgeeks.org/introduction-to-merkle-tree/#

Taustin. (n.d.). *Taustin/Spartan-Gold: A simplified blockchain-based cryptocurrency for experimentation*. GitHub. https://github.com/taustin/spartan-gold

What is block size? - bitstamp learn center. Learn Center. (n.d.). Retrieved from https://www.bitstamp.net/learn/crypto-101/what-is-block-size/