Bowdoin Coin

A Basic Implementation of a Transaction-Focused Crypto Currency Using a Blockchain Distributed Ledger

Evan Albers and Mustafa Aydogdu
Department of Computer Science
Bowdoin College

Abstract

Crypto currencies have experienced a meteoric rise in popularity in the past decade. What began as a fringe enclave of individuals distrusting of centralized authority has become a respected asset class, even achieving adoption within more traditional realms of finance. While crypto currencies serve a wide range of potential applications, from organizing smart contracts to even representing digital art, they all generally adhere to a fundamental structure that underpins both their decentralized nature, and their security: the blockchain. The blockchain both provides a unique example of many of the general challenges of distributed programming, as well as some of the more particular challenges of peer-to-peer systems.

Our system implements a basic currency using a blockchain distributed ledger. We do not offer support for more complicated constructs, such as smart contracts. BowdoinCoin's exclusive capability is to process BowdoinCoin transactions between key-pair wallets, in a manner that assures both the security and the validity of transactions. We believe BowdoinCoin to be highly scalable, and one of its outstanding features is that it does not require high computational power for the mining process thanks to the use of a system that we call "grace period". We hope that our implementation of BowdoinCoin will serve as a useful demonstration of the tradeoffs involved in designing and implementing a blockchain, even at the basic level of processing transactions.

1 Introduction

Crypto currencies are intended to provide a digital representation of something, be it monetary value or a unique token, that cannot be altered by any agent that is not in control of the given asset. Crypto currencies ensure this property with a series of cryptographic properties that undergird most crypto currencies. The foremost tool of secure encryption is the SHA256 hashing algorithm. SHA256 is designed to provide a unique output for any input such that changing a single bit of the input results in a completely different output. It is impossible to "crack" the hash by working backward from some given hash-value to determine the input string that originally generated it. Therefore, hashing can serve as a monitoring system; if some string is stored in a public area, and each user maintains a hash of the original string, it is possible to tell if some actor, maliciously or otherwise, alters the string. If one were to compare the hash of the current value with the original saved hash, and they were to differ, then the original string value must have been altered. Otherwise, applying the hash function to the string would have returned the same value. Hashing is used within generalized crypto-currencies in three significant ways: securing the chain itself, authenticating transactions, and as a foundation of the mining process.

1.1 The Blockchain

The blockchain itself is a method of organizing and ensuring the validity of certain chunks of data: blocks. A block is a subset of data stored in a distributed ledger. Each block stores a small amount of metadata about itself in its block header. This includes the number of transactions within the block, its "height" in the chain, as well as the hash value of the previous block. The hash value of each preceding block is what "chains" each block together. Each block has a unique value that results from hashing the block. If an agent were to alter the block, they would alter its hash value. Because each block also contains the hash value of the preceding block, altering one block changes the hash value of the next as well. This propagates down the chain. This property alone protects blocks early in the chain; however, because the chain is stored on every node in the network, a malicious agent would have to alter every descendant block on each node in order to effect their change. This task becomes impressively more difficult as the network increases in size. Hashing also ensures that malicious actors cannot forge transactions from accounts they do not control.

1.2 Transaction Security

Hashing, in conjunction with RSA keypair encryption, ensures that transactions cannot be forged by any party without access to the private key of the sender. When a sender wishes to initiate a transaction, they generate a string that represents the transaction. They then hash the string, and "sign" it, by encrypting it with their private key. They send the signed string (representing the hashed transaction) to nodes in the network, alongwith the raw transaction string. The nodes then use the sender's public key to decrypt their signature. They also hash the raw transaction. If the hashed transaction matches the decrypted signature, then the transaction must have been signed by the sender with the corresponding public key. Therefore, it is a valid transaction. This process is demonstrated in Figure 1. It is important to note that this does not prevent some malicious actor from stealing a wallet's private key in order to generate transactions. A user is responsible for protecting their own private key. Finally, hashing also provides a method by which the network can help synchronize and reward nodes in the form of mining.

Figure 1.

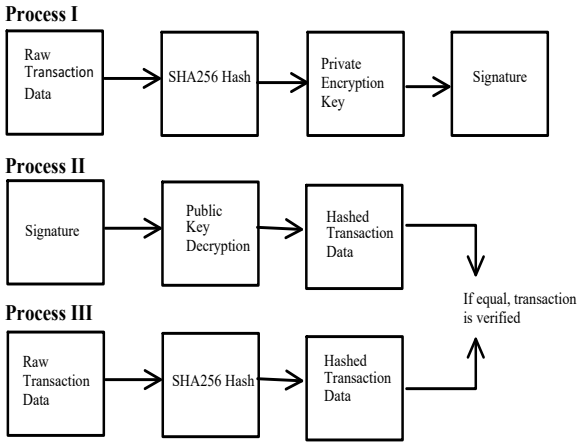

Figure 1 displays the three key processes involved in signing and verifying transactions. Process I is the sender node's signing of the transaction. Processes II and III describe the method by which the recipient node decrypts the signature and verifies it using the raw transaction data.

1.3 Mining

The very same hash function that undergirds the security of cryptocurrencies also helps serve as a rule set for "mining." Mining serves a handful of roles for cryptocurrencies. It both helps reward miners and synchronize the generation and addition of blocks to the network. Concurrent blocks pose an issue for blockchains. If there are two blocks created concurrently, one must be deemed the "real" block, and added to the chain; the other must be discarded. Mining is a way of organizing

the behavior of blocks; blocks can only generate a block containing their "unmined" transactions – transactions not yet committed to a block – when they complete a certain amount of work. This work is generally constructed to limit the issuance of blocks across the network to a predetermined rate. It also serves to help verify that a node has the right to issue a node. Mining also allows the network to reward nodes for staying online – an issue in a peer-to-peer system – and completing the work of processing transactions. Typically, nodes receive rewards based on the difficulty of their mining block. All these features are implemented within BowdoinCoin, albeit with some modification.

2 Implementation

Our implementation of BowdoinCoin can be broken down into three primary components: Node functionality, network functionality, and cryptocurrency functionality. Each portion presented significant challenges of different varieties.

2.1 Node Functionality

Each BowdoinCoin node is capable of a set of basic functions that enable the formation of the network, and transactions on the network. Each node maintains a list of connections. Each individual connection is referred to as a "neighbor." Each node also has an arbitrary constant representing the "maximum neighbors" that it ought to have. This is to prevent nodes from making too many connections. Each node also keeps a list of pending transactions, which contains all the transactions that a node has received and verified, but not yet witnessed within a block. Finally, blocks also maintain a "top block" that represents the top block of the blockchain. The blockchain itself is a linked list of blocks, in which each contains both the next block and the previous block.

There is also an ideal number of neighbors for a node, which needs to be less than the maximum number of neighbors. A node accepts incoming connection requests up to the ideal number; however, it makes requests to connect to other nodes up to the maximum number of neighbors. Having two different parameters for the number of neighbors and having this gap between them ensure that the network never gets into a closed-circuit position where all nodes are saturated with the maximum number of neighbors and no node can make a connection to a new node. To portray this problem, imagine that the maximum number of neighbors is 4 and there is no ideal number. As soon as there are 5 nodes in the network, every node will establish connection with all other ones and will reach the maximum. No new node will be able to join the network anymore. We are going to touch upon the use of these numbers in 2.2 again.
We are currently using a maximum number of 12 and an ideal number of 8 in our system.

2.2 Network Functionality

BowdoinCoin is fundamentally a peer-to-peer network that facilitates transactions. We decided to implement BowdoinCoin using Java. Our network functionality is provided using socket connections, which allowed for a more straightforward implementation of the currency versus managing server-client relationships in with remote procedure calls. It also granted us the freedom to conceive of BowdoinCoin as a protocol, rather than a rigid piece of software, much like commercial cryptocurrencies such as Bitcoin. Implementing BowdoinCoin as a protocol as well as a specific piece of software highlights the largely theoretical nature of the currency. Some agent could write their own BowdoinCoin node software, and so long as it followed the message protocol of the network, the blocks it generates could still be accepted and added to the chain. This could

occur without compromising the security or validity of the network. In addition to determining which language to use when implementing the network, deciding *how* to implement it also presented significant challenges. BowdoinCoin suffers from similar problems to many peer-to-peer networks. Users are not stable, and therefore the network itself can be difficult to balance. Furthermore, it must be able to gracefully increase and decrease functionality based on the number of nodes in the network. Finally, nodes need to be able to join the network, and manage their own connections. Each node keeps track of their "neighbor" nodes (nodes to which it maintains connections) in a list of connections. Each connection is represented by a connection object. Connection objects extend the thread class, and constantly monitor their connection with their neighbor, handling the processing and sending of messages in the connection. We implement the joining of nodes in a unique fashion, with the function "joinNode." joinNode is called from a node already in the network; it is passed the public IP of a machine that wishes to join the network. In this sense, BowdoinCoin is "invite only" – if a node does not already know a node in the network, it is not possible to join. Upon joining a node, the new node sends a "populateNeighbors" message to the in-network node. This message contains the public IP of the new node and a counter that represents how many times the message has been repeated. Upon receiving a populateNeighbors message, a node checks if it has the maximum number of neighbors (an arbitrary value meant to limit the number of connections a node has); if it does not, then it will immediately connect to the IP given in the message. If it already has the maximum number of neighbors, then it will not connect. In either case, it will also check the value of the repeat-counter. If the counter is less than the maximum repeat number, then it will broadcast the message to all its neighbors, if not, then it will not. This process is generally conveyed in Figure 2. The counter is needed because we don't need this message to circulate in the network until it reaches every node. It should be enough to circulate this message for a fixed number of hops and the new node should reach the ideal number of neighbors.

Figure 2.

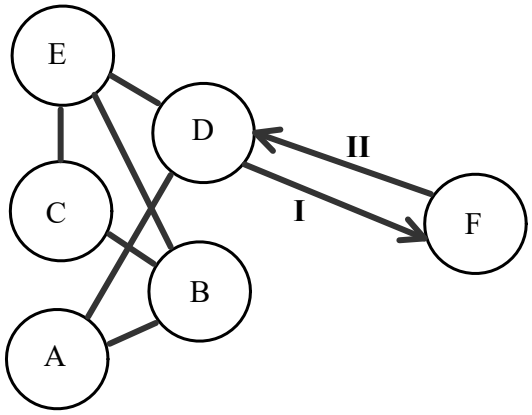

Figure 2 offers a view of a possible BowdoinCoin network. Some nodes are connected to other nodes, but not every node is connected to every other. Node F wishes to join the network. Therefore, node D connects to F after calling joinNode, represented by arrow I. F then sends a populateNeighbors message to node D, represented by arrow II. This message then is broadcast again by D, and so on by A and E, by which it eventually reaches C and B.

Node failure poses very little issue in BowdoinCoin. There is no node hierarchy, ideally every node is identical to every other, although in practice the transactions stored on each node may vary slightly due to network latency. If a node fails, then it is simply removed from the neighbor list of its respective neighbors. This is the extent of BowdoinCoin's network functionality. While reasonably straightforward, it provides a crucial platform for the execution of BowdoinCoin's crypto currency functionality. Implementing currency functionality, and by extension, BowdoinCoin's blockchain,

involved important tradeoffs regarding network performance and security.

2.2 Currency Functionality

Physical currency has properties that ensure its usefulness as a store of value. Dollars cannot be conjured out of thin air, nor can they be spent more than once. Therefore, for BowdoinCoin to be useful, it must be able to prevent both counterfeiting and double spending. The central axiom that guarantees significant portions of BowdoinCoin's transaction security is our definition of a confirmed transaction. A transaction is confirmed (i.e., can count towards a wallet's balance) if and only if it has been verified and added to the blockchain. This provides a useful framework for understanding how BowdoinCoin provides currency functionality: transaction verification, and block generation and distribution.

2.3.1 Transaction Verification

Transactions are verified using the general process described in section 1.2. Each node verifies every transaction that it receives. If a node is unable to verify a transaction, it will not broadcast it to the rest of the network. This ensures that invalid transactions will be stopped as soon as they cannot be verified. A transaction must also be possible within the context of a user's present balance. If the user does not have the requisite funds in associated with their private key, then they cannot complete the transaction. This criterion extends to the unconfirmed transactions that a node stores while mining. If a node receives two transactions that are conflicting – a user might try to spend more than their wallet contains – then it will privilege the earlier transaction. Block generation and distribution also significantly relates to the verification of individual transactions.

2.3.2 Block Management

A block is generated whenever some node happens to chance upon a hash value that contains the at least as many leading zeros as the mining difficulty. When a block is generated, each transaction is checked for consistency and validity; no wallet may spend more than its most recent balance on the chain. Earlier transactions, sorted by timestamp, are privileged to later ones. If there are invalid transactions within the block, possibly due to network latency issues, they are removed on a last in, first out basis. Blocks are broadcast to the network as they are mined. BowdoinCoin currently has no mechanism to resolve concurrent block creation. We rely largely on the fact that a sufficiently high difficulty level will severely decrease the likelihood of concurrent block, although this also presents a constraint on transaction-confirmation performance. There are commercial currency implementations that resolve this issue, although the complexity involved in implementing such solutions placed them beyond the scope of BowdoinCoin's implementation. There are a handful of avenues for evaluating BowdoinCoin's performance, ranging from block creation to mining speed.

3 Evaluation

Significant aspects of BowdoinCoin's performance are binary in nature; either the currency is secure, or it is not. However, there is still some scope to evaluate the performance of its constituent functions. Our evaluation helps illustrate key tradeoffs involved in determining block size, and the tradeoff between maximizing transaction performance and minimizing concurrent block creation.

We decided to evaluate the performance of BowdoinCoin's buildBlock() function to glean insight about the ideal size of a block.

Figure 3.

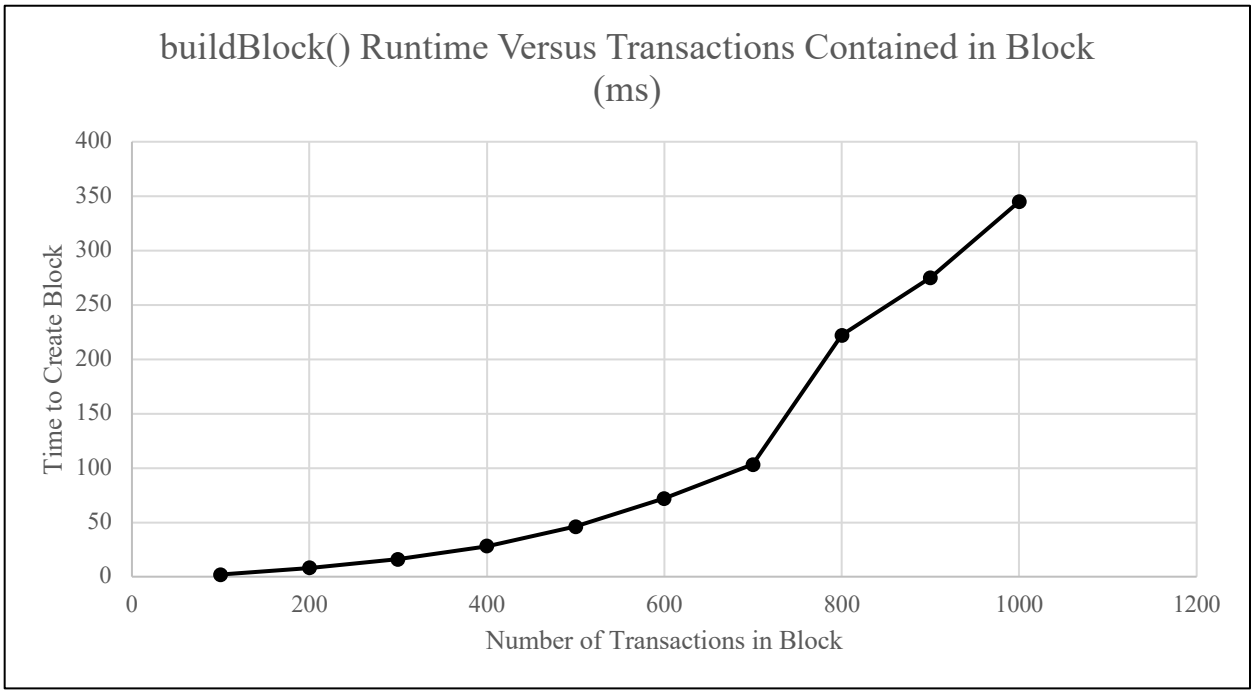

Figure 3 displays the runtime of Node.java's buildBlock() method. The data was collected by writing a short program to generate transactions, and then construct a block. Each data point represents the average block creation time of 100 calls to buildBlock() for the given number of transactions.

The "cleaning" (removal) of contradictory transactions and construction of each block is not linear and is a function of the number of transactions within the block. To evaluate the performance of the function, we wrote a script that sends a given number of transactions between two nodes, and then calculates the runtime of creating the block containing the generated transactions. This process was repeated 100 times for each increment of 100 transactions, to achieve a representative average of the running time.

Figure 3 displays the results of this experiment. As would be reasonably expected, increasing the number of transactions in a block also increases the time required to construct the block. However, the relationship is not linear. The time required increases at a greater rate as the number of transactions grows. This is likely due the cumbersome nature of cleaning each block. This process is necessary to maintain security but is costly in terms of performance. Figure 3 would suggest that in order to maximize performance during block creation, the number of transactions per block should be kept low. However, if there is a low number of transactions per block, either blocks must be mined frequently, or transaction-confirmation performance will suffer. This tradeoff in part motivated our second evaluation regarding mining difficulty.

Our second experiment demonstrates the relationship between the time taken to mine a given block, and the mining difficulty – the number of leading zeros required to satisfy the hash requirement. We wrote a short program that mines hash values for a given difficulty value, finding 100 values for each level of difficulty, in an effort to determine a reasonably representative average of the

Figure 4.

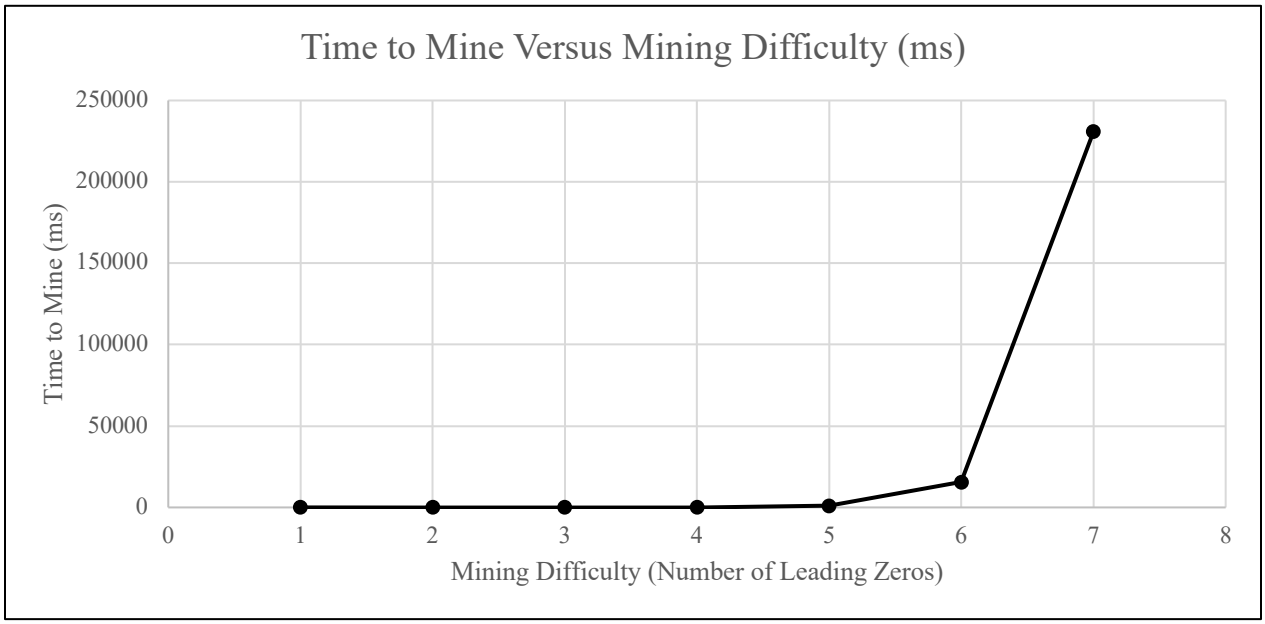

Figure 4 displays the time required to mine a block at a given level of mining difficulty. Each data point represents the average runtime of 100 miner executions.

runtime for a given level of difficulty. It is important to note that the specific runtimes will vary across machines. This evaluation merely displays the relationship between the time taken to mine, and the difficulty level. This relationship should hold across all machines, although it is possible that there is some mathematical optimization that could alter this performance. We do not consider such a possibility.

Figure 4 presents the results of our mining evaluation. The first five data points are 1, 2, 8, 67, and 945 milliseconds. The explosion in runtime is so extreme as to render the graph nearly unreadable. The runtimes can be reasonably interpreted as a block-mining rate; each time a hash is discovered, a block will be created and broadcast to the network. This also demonstrates a key tradeoff. Decreasing the mining difficulty will increase the rate of block discovery, which decreases the confirmation time of transactions. However, it also substantially increases the likelihood of concurrent block discovery, especially as the number of actively mining nodes in the network increases. Consider mining difficulty 5. On a network of machines of a comparable power to the one on which the evaluation was conducted, the rate of concurrent block discovery would vary dramatically with the number of nodes in the network. It took about one second for our machine to discover a valid hash. If five machines were mining, it would take a fifth of a second, on average, over the long run. A concurrent discovery, then, is five times as likely, even at the same mining difficulty. A similar relationship could hold for nodes running multiple mining threads, so long as increasing the number of threads still offers returns in computing power.

Finally, we also wanted to evaluate the performance of an individual node's ability to create transactions. MakeTransaction is a synchronized method to prevent some user from creating multiple threads to double spend coins on an individual node. While this is checked for when creating a block, it would create issues in the interim, as users try to check their balances before they have been

Figure 5.

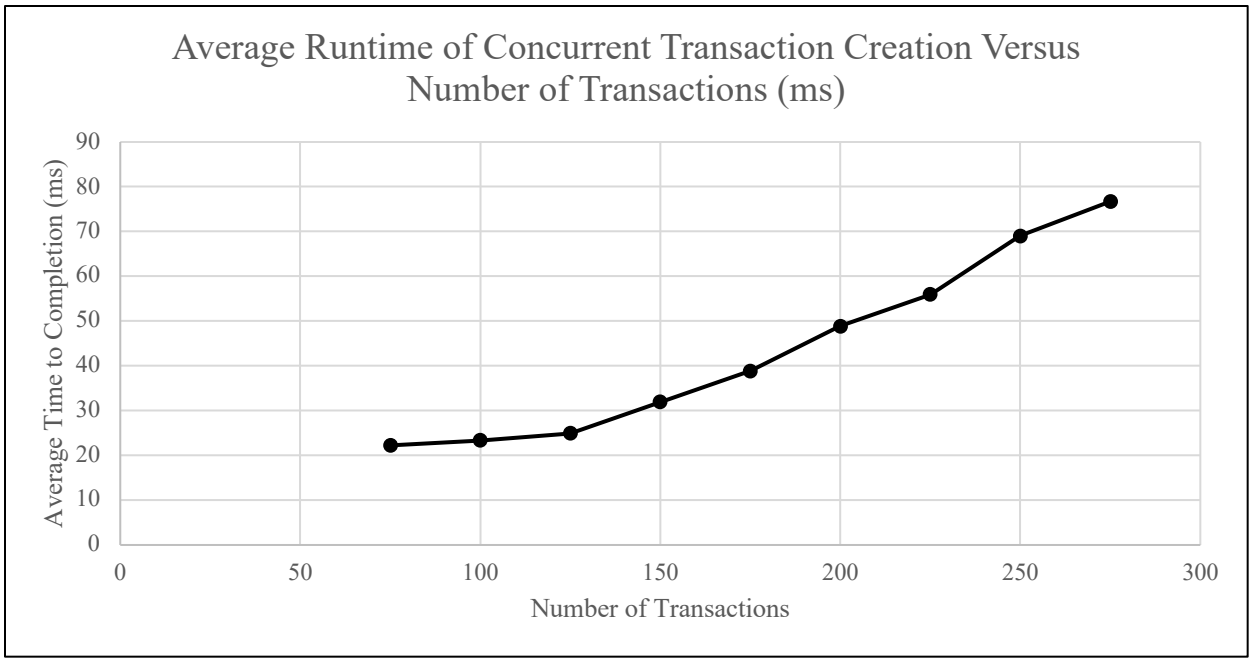

Figure 5 demonstrates the results of our evaluation of concurrent transaction creation. Five nodes each concurrently executed a given number of MakeTransaction calls, and divided the total runtime by the number of calls to determine the average runtime per call.

committed to the blockchain. To test the performance of MakeTransaction, we ran a short client that created five threads each carrying out some number of transactions. We then calculated the average runtime of creating each transaction. Figure 4 displays the results of this evaluation. The increase in average runtime per MakeTransaction call is not quite linear, indicating the initial overhead stemming from the synchronization of the method.

4 Conclusion

Implementing BowdoinCoin has presented an array of difficulties. We were faced with a series of decisions regarding BowdoinCoin's performance as a currency, and its overall security. In almost every case, we chose to prioritize security over performance, as this seemed crucial to the coin's fundamental usefulness as an asset.

There are a handful of functionalities that we would like to provide support for in the future, to further demonstrate the longevity of the system and its functionality. Possible avenues for future improvement include downloading blocks to store them on the local machine, and implementing a method by which nodes can bring other nodes "up to speed" by forwarding their current chains to new nodes. However, despite these potential avenues for improvement, BowdoinCoin still provides solid currency functionality with reasonable performance, and strong security.