
AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Bachelor report in Computer Science

# Aucoin

## a distributed cryptocurrency

*Authors*:

Casper V. Kristensen - 201509411
Magnus Meng Mortensen - 201506764

*Supervisor*:

Claudio Orlandi

15th June 2018

Academic Year 2017/2018

## Abstract

Cryptocurrencies allow purely peer-to-peer online payments without a trusted third party. In this project, we develop a cryptocurrency system based on the Bitcoin protocol. The solution distinguishes itself from others by disincentivising centralisation and improving security through a simple architecture, preventing attacks conceivable in prominent systems today. The only centralised component of the design is shown to be of minor significance for the operation of the system, and algorithms that draws inspiration from the field of engineering are demonstrated to provide an almost constant block time, improving the stability of the system. The result is a scalable and reliable cryptocurrency that is almost completely decentralised.

# Contents

# 1 Introduction

This report seeks to find the necessities of a cryptocurrency by studying the original implementation; Bitcoin. The report documents the process of implementing and designing a simple alternative to Bitcoin, namely, the Aucoin cryptocurrency. The main focus is to make Aucoin a *truly* distributed cryptocurrency, that is testable, scalable, and does not rely on any centralised services.

## 1.1 References and Literature

The references used in this work can be found in the References section at the end of the report. Before continuing, we want to discuss the academic challenges of finding literature in the area of cryptocurrencies. Due to the nature of cryptocurrencies, not many academic writings provide the required level of detail to develop, implement, and document a cryptocurrency. The main source of literature used in the report are found on online discussion boards and forums, where users and developers alike discuss both high-level concepts and low-level implementation details. It is important to note that while many references are from these online forums, the vast majority of information was extracted from posts authored by core developers or stakeholders of prominent cryptocurrencies like Bitcoin. For this reason, we do not see an immediate problem using these resources as a source for technical and conceptual choices regarding Aucoin.

# 2 Introduction To Blockchains

A blockchain is essentially a slow distributed database. It consists of several components which we will briefly introduce:

**Network** The blockchain functions in the context of a network of users, that can be either public or private. A private network is typically focused on removing market friction in different sectors of industries, and allows for fine grained access control. A public network, on the other hand, is especially suited for cryptocurrencies, as it allows anyone to participate in the network.

**Nodes** We will in this report address users in the network simply as *nodes*, though in reality there can be many different type users of a blockchain. When we refer to *nodes*, we indicate a physical instance capable of participating in the blockchain network.

**Shared ledger** The core of the blockchain is a shared ledger, which is the component responsible for collecting blocks of transactions in a consistent immutable state.

## 2.1 Types and Differences

Before beginning, let us first introduce the different types of blockchains. We will cover permission-less and permission-based blockchains. We then discuss

why we choose to focus on implementing a public permission-less blockchain.

### 2.1.1 Permission-less vs. Permission-based Blockchains

Blockchains enhance trust between networks. It does so by being a shared distributed ledger which provides the means of recording, applying, and tracking transactions in a network. One might think of blockchain as an operating system with several applications, like Bitcoin, Ethereum, Litecoin, etc. When we speak of blockchains, we easily confuse two types of access control; permission-less and permission-based. Both are decentralised peer-to-peer networks, where each node maintains a shared ledger of append-only transactions. By applying a consensus protocol to the ledger, the nodes maintain a collection of valid transactions guaranteeing immutability of the ledger. The sole exception between the two types of access control is whether it enables the creation of members-only networks and ability to prove membership. Through IDs and permissions, permission-based networks offer management of the level of detail available for specific users when browsing transactions. It also allows improved auditability by having a shared ledger serving as a single source of truth. Though sharing this property with the permission-less blockchain, it offers more fine grained control. The performance of the permission-less blockchain suffers at scale, as the consistency, as well as the immutability of the ledger, depend solely on computationally expensive tasks. A permission-based blockchain does not depend on such tasks to be practical, and can therefore scale more easily. For this reason, permission-based blockchains are generally preferred in the enterprise [11, p. 6-18].

### 2.1.2 A Public Permission-less Blockchain

One might argue that a private permission-based blockchain is suitable for problems considering enterprise solutions. But what we intend with this project is to develop a blockchain implementing a cryptocurrency called *Aucoin*, and study the aspects of it through experiments. Therefore, we choose to use a public permission-less blockchain. This implies that any user will be able to connect to the system, publish new transactions, as well as engage in the consensus protocol with equal limitations. This has some challenges that need to be considered: We will need a stable and scalable interface for distributing the shared ledger, publishing transactions, and keeping everything consistent. Also, we need a consensus protocol that guarantees the immutability of the shared ledger.

### 2.1.3 Consensus

In networks where blockchain users are not known, we cannot trust them. The solution is to build the security of the system around not needing to trust them. This is the job of the consensus protocol; it ensures that every transaction appended to the blockchain is valid. It is a common agreement that every user of the blockchain must follow.

**Proof of Work** Originally proposed by Satoshi Nakamoto in 2008, the Proof of Work (PoW) consensus protocol, is a computationally hard task that needs to be solved in order to push new transactions to the blockchain. The task is to find a value, such that when hashed with a predefined hash function, it is below a given target. This value can easily be checked with a single hash. Usually, as in the case of Bitcoin, PoW is accomplished using blocks of transactions. That way, if an adversary wants to change transactions in the blockchain, it is impossible without having to redo the work of all the preceding blocks. This is an extremely expensive task, virtually rendering the blockchain immutable [16].

**Proof of Stake** PoW has one major downside; it requires constant computational power to ensure the security of the network, creating a massive waste of resources. The common alternative to PoW is Proof of Stake (PoS). PoS applies the user's aggregated transaction value or balance, to determine who is selected to create a new block. Whoever has the highest stake in the network, will have the greatest chance of creating a new block. PoS builds on the assumption that users who have more to lose by attacking the network will generally refrain from doing so [11].

The reader will later discover that Aucoin implements PoW as part of the consensus protocol. The reason is that PoW is one of the simplest and most well-documented consensus protocols, and it aligns nicely with the time span and expectations of a bachelor project. The theoretical aspects of PoW includes protection through signatures and computations with hash functions, which we detail later.

# 3 Introduction to Aucoin

Aucoin is a cryptocurrency that implements a pubic permission-less blockchain. It is build on simplifications of the Bitcoin protocol with the purpose of researching how to build a scalable and reliable cryptocurrency system. The main focus is to implement and experiment upon a modular blockchain that disincentivises mining pools and offers stable difficulty adjustment to arrive at an average block mining time of 60 seconds. Aucoin is modularised in 8 modules which are all outlined with dependencies in figure 3.1.

# 4 Transactions

Transactions allow users to spend coins by transferring ownership from one user to another. In this system, users are identified by their public key, by which the ownership of the coins is reassigned to the receiver's public key. A transaction contains at least one *input* and one *output*; Each input references the output of a previous transaction, thereby spending it. The outputs assign these coins to one or more receivers. In this way, transactions form the vertices in a graph with edges between inputs and outputs, representing the change of ownership [16].

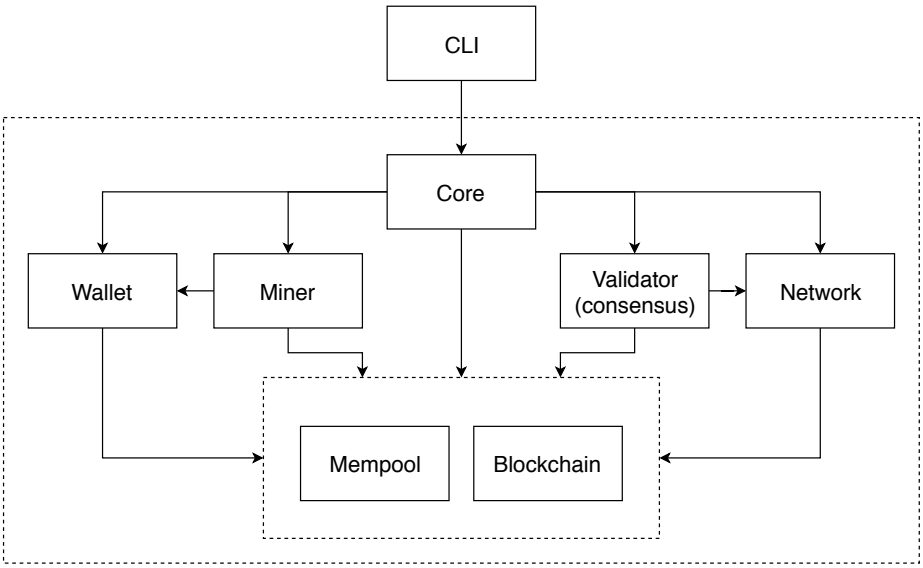


**Figure 3.1:** A diagram illustrating the relationship of the 8 modules of Aucoin. Notice the 3-layered architecture; outer interface layer with CLI, a core layer, and a storage layer.

The first transaction in this chain, however, does not reassign coins from a previous output. This special transaction, called the *coinbase transaction*, generates new coins and awards them to miners; this is how new coins are introduced into the system. Coinbase transactions are very different from standard transactions, and as such they are discussed separately in section 6.1 while the mining process itself is detailed in section 6.

All transaction outputs that have not yet been referenced by any inputs are called *unspent transaction outputs* (UTXOs). The sum over the value of all UTXOs for a given public key is denoted its *balance*. In this way, transactions not only allow transferring coins, but actually define the notion of owning coins, by which ownership entails controlling the private key associated with a public key of an UTXO.

An illustration of a chain of transactions can be seen in figure 4.1. Note that there are no connections between inputs and outputs within a transaction; this means that it is impossible to define and trace a single "coin".

## 4.1 Scripts

In Bitcoin, inputs reference previous outputs by the *previous transaction* and *index* fields. These fields contain the hash, or unique identifier, of the previous transaction the output is contained in, as well as its position within the output vector, respectively. Outputs consists of a *value*, denoting the number of coins it will be worth when spent. Both inputs and outputs have a *script*; when validating the transaction, the two are combined and evaluated. If the script returns true, the input is authorised to spend the output and the transaction is valid [43]. Though not Turing-complete, the Bitcoin scripting system allows the construction of many different types of transactions. For example, it is possible

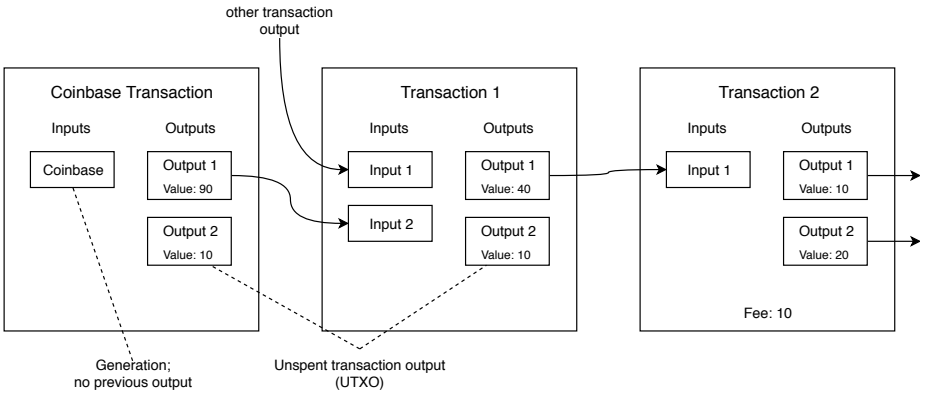


**Figure 4.1:** A chain of transactions. The inputs of the coinbase transaction does not reference a previous output, and the outputs which have not yet been referenced by any input are unspent. The difference between the implied value of all inputs and the value of all outputs in a transaction is the transaction fee.

to construct an output in such a way that it can be spent by anyone calculating the solution to any pure function, such as a hash or encryption function [33].

In the first release of the Bitcoin protocol, coins were paid directly to the user's public key. That is, the *script* field contained a script that would return true for any person able to produce a signature using the corresponding private key. Scripts of this type are known as *pay-to-pubkey.* Today, the most typical Bitcoin transaction is the *pay-to-pubkey-hash* transaction [13]. The output scripts of these transactions contain a hash or "address" of a public key. To redeem the output, the receiver must construct an input script which both provides a signature using the corresponding private key, but also proves that the public key indeed hashes to the receiving address [42].

The obvious disadvantage of pay-to-pubkey is that the sender needs to know the public key of the receiver in advance. Conversely, pay-to-pubkey-hash allows users to send coins to shorter, more memorable addresses. Moreover, if the user never reuses an address, they do not have to publish their public key until the coins are spent. Allowing the public key to be withheld is crucial for the security of the system, since quantum computers are able to derive the private key from an ECDSA public key [21]. By utilising pay-to-pubkey-hash, however, the security relies not only on the ECDSA scheme, but also the hash function, which is considered to be relatively secure against attack by quantum computers [2].

## 4.2 Signing Transactions

The signature in the input scripts of a *pay-to-pubkey-hash* Bitcoin transaction provides authenticity that the receiver of the referenced output is the one spending it [42]. Each signature contains a SIGHASH flag, specifying which parts of the transaction it signs [37]. This not only protects the transaction from modification, but also allows the creation of transactions constructed using a combination of inputs signed by different users, where each party signs only

a part of whole transaction, allowing other parts to be changed without their involvement [33].

Bitcoin currently supports three base SIGHASH flags [19]:

**SIGHASH_ALL** This is the default flag, which indicates that the *previous transaction* and *index* fields of all inputs as well as the *value* and *script* fields of all outputs are signed. Additional transaction metadata like the header is also signed, effectively protecting everything except the input scripts from modification [19]. Note that the input scripts are not included in the signature, since that would involve constructing a signature which signs itself [33].

**SIGHASH_NONE** Signs all of the inputs like SIGHASH_ALL, but none of the outputs. This allows anyone to change where the coins are sent [19].

**SIGHASH_SINGLE** This flag is similar to SIGHASH_ALL, except that the only output which is signed is the one with the same index as this input. This enables others to append outputs, as long as the original single output is unaltered [19].

In addition to the base flags, Bitcoin also supports a modifier called SIGHASH_ANYONECANPAY. This additional flag causes only the single input to be signed, and can be combined with any of the above, creating three new combined types [33]. For example, the combined flag SIGHASH_ALL | SIGHASH_ANYONECANPAY signs all of the outputs, but only this one input, allowing anyone to add additional inputs. This could be used to organise crowdfunding campaigns; in this case, the transaction is only valid if enough inputs are added to cover the value of the outputs, so nothing is transferred if the campaign does not reach its goal [19].

## 4.3 Transactions in Aucoin

Unfortunately, implementing a scripting language similar to the one found in Bitcoin is outside the scope of this project. Instead, we have focused on supporting the most commonly used transaction – the *pay-to-pubkey-hash* transaction – as well as the default SIGHASH flag: SIGHASH_ALL. As shown in figure 4.2, these constraints simplify the structure of transactions significantly: Instead of an output script, outputs simply delegate the value to an address as defined by definition 4.1.

**Definition 4.1** (Address)**.** The Aucoin address of a public key $p_k$ is defined as

$$address(p_k) = \text{sha256}(p_k)$$

The validation engine replicates the steps performed by a *pay-to-pubkey-hash* signature script by checking the validity of the signature using the accompanying public key according to the SIGHASH_ALL rules. The output script is replaced with a check that verifies that the public key matches the address of the referenced output, as defined in definition 4.1. Together, these checks substitute the scripting language found in Bitcoin, but it must be stressed that they only compose a small subset of the validation checks necessary to ensure the security of the

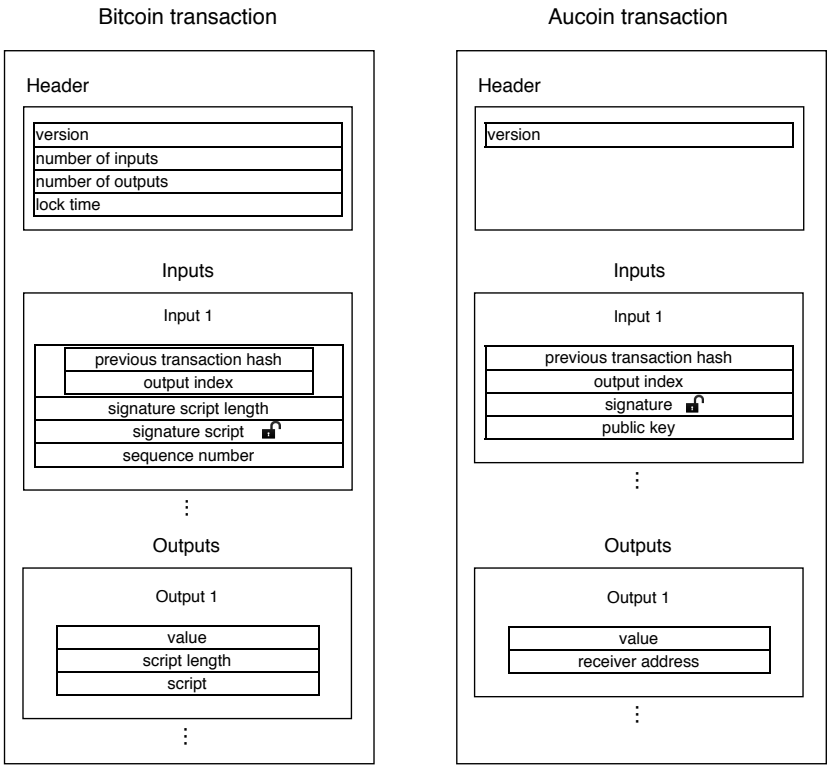


**Figure 4.2:** The difference between a Bitcoin transaction [20] and an Aucoin transaction. The open padlock represents an unsigned field.

system. For example, to prevent double spending the validator also checks that each output is referenced by an input at most once.

## 4.4 Transaction Malleability

As stated previously, the signature scripts in a Bitcoin transaction are not part of the data that is signed, since doing so would result in a circular dependency. The same applies to signatures in an Aucoin transactions, as symbolised with an open padlock in figure 4.2. In both Bitcoin and Aucoin, however, the script and signature, respectively, contributes to the transaction's hash, which is used to uniquely identify it when referencing previous outputs from an input [19]. In the case of Bitcoin, the flexibility of the signature scripts allow an adversary to make non-functional modifications to a transaction without rendering it invalid, for example by adding a NOP opcode to the script [42, 19]. It should be noted that this does not change what inputs the transaction uses, nor what outputs it pays, so the coins will still go to their intended recipient [45]. It does, however, mean that the computed hash of the transaction changes, which causes it to have a different unique identifier than the creator expected.

As a practical example, consider an attacker requesting a withdrawal from a Bitcoin exchange. The exchange would create a transaction and publish it

on the network. By utilising transaction malleability, an attacker might then be able to trick poorly implemented payment tracking, since it may appear that the transaction has disappeared from the network. This would cause the exchange to believe that the transaction has failed, crediting the amount back to the attacker's account, effectively doubling the balance [10]. Because of this possibility, best practices dictate that Bitcoin transactions should be tracked by the transaction outputs as opposed to its hash, and that reissued transactions spend the same outputs as the previous one, invalidating the original one [19].

Interestingly, the simplicity of Aucoin transactions solve the malleability problem; since the only unsigned field is the signature, no modifications can be made without invalidating it.

## 4.5 Fee

For the transaction to be considered valid, the sum of output values must be less than or equal to the sum of output values referenced by the inputs. The difference between these two values is called the *transaction fee* and is formally defined as follows:

**Definition 4.2** (Transaction fee)**.** For a transaction $tx$ with input vector $v_{in}$ of inputs and output vector $v_{out}$ of outputs, the transaction fee is defined as follows [18]:

$$\text{fee}(tx) = \sum_{\text{input} \in v_{in}} \text{input.output.value} - \sum_{\text{output} \in v_{out}} \text{output.value}$$

Note that an input does not define a value. Instead, the value of the output it references must be retrieved to calculate the fee. According to the consensus rules, the fee must be non-negative in order for the transaction to be considered valid. The fee of a given transaction is paid to the miner who includes it in a block; the specific process is examined in section 6.

Due to the above definition, the referenced outputs are always spent in their entirety. This means that the sender must include an additional output sending excess coins back to themselves to avoid paying the difference in fees. This technique is known as *returning the change*, analogous to the change received when paying with cash [16].

### 4.5.1 Privacy and security considerations

In Bitcoin, it is recommended [32] to generate a fresh address for every transaction that requires change. This is due to the fact that all Bitcoin transactions are public, allowing anyone to monitor transaction amounts. In theory, generating a fresh change address effectively conceals which of the transaction outputs are intended for the actual receiver and which is the change [22]. In practice, however, the user may have to use multiple previous change outputs to cover the amount of a transaction in the future. This allows an observer to determine with high confidence that the addresses are under the control of a single user, deducing their status as change addresses. Thus, the intended recipient and amount of previous transactions is only concealed as long as the change outputs

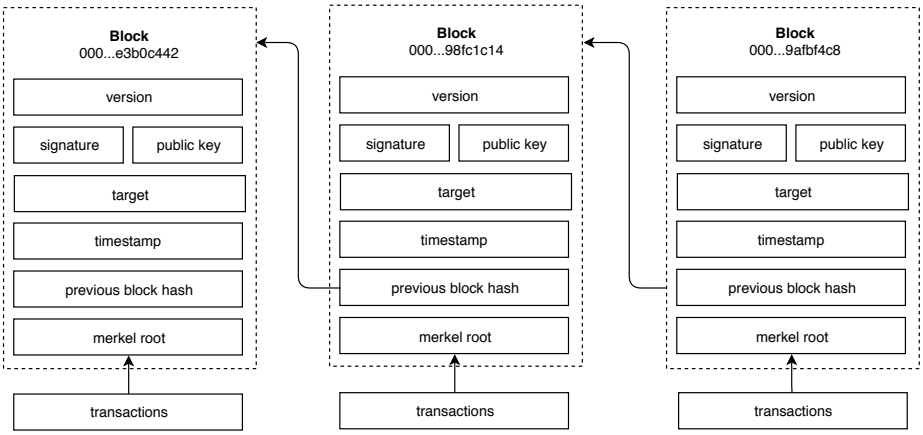


**Figure 5.1:** The structure of an Aucoin block. The transactions contribute to the hash indirectly through the Merkle root.

cannot be correlated. Over time, however, the change technique will cause the user's coins to disperse into UTXOs of increasingly smaller value, until at some point it becomes impossible to construct any transactions without spending multiple change outputs in conjunction [28].

Another consideration is that reusing the same change address would reveal the public key upon spending any of the outputs. This decreases the security of the sytem in a post-quantum world, as per the discussion in section 4.1.

# 5 Blocks

In the previous section, we explored how the inputs and outputs of transactions form a chain, redefining ownership of coins. So far nothing prevents the owner of an output from transferring it to multiple receivers without their knowledge. This is the infamous double-spending problem, which was solved by Satoshi Nakamoto in 2007 with the introduction of the blockchain [16].

The blockchain works as a shared database of all previous transactions, allowing users to agree on a single history. In this way, users are aware of all transactions performed in the system, by which it is possible to confirm the absence of double-spending transactions. The blockchain is constructed by collecting transaction into blocks, which form a chain by a process similar to that of a linked list [16]. We denote the list of contained transactions the "body" of the block. Furthermore, the block also includes a header with additional metadata, and they are uniquely identified by the hash of this header, which is constructed in such a way that it depends on the list of transactions. For this reason, the hash encapsulates all the data in the block [20, 39]. The contents of an Aucoin block is visualised in figure 5.1 and described next.

**Version number** The version number is a header field consisting of an integer that indicates which block validation consensus rules to follow. It allows the introduction of changes to the protocol that are incompatible with previous

versions. It is important that this field is part of the data that constitutes the block's hash to prevent an adversary from altering it, causing the nodes to apply a weaker set of validation rules to it.

**Previous block's hash** This header field contains the hash value of the previous block this one builds upon. The field is responsible for connecting the blocks into the block tree that is commonly known as the blockchain. In this way, the block tree consists of a number blocks organised into branches. The consensus rules ensures that exactly one of these branches is regarded as the *main* branch, thereby determining the blocks that make up Aucoin's public ledger. The algorithm responsible for this is described in section 9.2.

Since each block header contains the hash of the previous one, no block can be modified without also modifying every subsequent block that builds upon it. In this way, the amount of work needed to modify a particular block in the branch increases with every new block added to it, thus protecting the blockchain from tampering. The consensus rules ensure modifying a block in the blockchain requires on average as much computing power as the entire network expended between the time of the original block and the present time. This is only feasible for an adversary controlling a majority of the network's computing power and is what is known as a *51 percent attack* [38].

**Merkle Root Hash** The Merkle root is a single hash derived from the hashes of all transactions contained within the block, thereby ensuring that none of the transactions can be modified without modifying the block header. As the body of the block consists solely of the list of transactions, the Merkle root hash ensures that the block's hash encapsulates all the data it consists of.

**Definition 5.1** (Merkle Tree)**.** A Merkle tree is a complete binary tree equipped with a function *hash*, conforming to the requirements set forth by the common definition of a cryptographic hash function. For the two child nodes $n_{left}$ and $n_{right}$, of any interior node, $n_{parent}$, the assignment $\Phi$ is required to satisfy

$$\Phi(n_{parent}) = hash(\Phi(n_{left})||\Phi(n_{right}))$$

where || means concatenation [26].

In our context, the bottom row of the Merkle tree is constructed using the hashes of the transactions in the block. Each pair of leaves is concatenated together and hashed to form a second row of hashes in accordance with definition 5.1. If there are an odd number of nodes in any row of the tree, the last node is paired with itself, i.e. $\Phi(n_{parent}) = hash(\Phi(n_{left})||\Phi(n_{left}))$. The process repeats, creating additional rows, until only one hash remains; the Merkle root hash [20, 39]. An example construction of a Merkle tree can be seen in figure 5.2.

One of the benefits of using a Merkle tree rather than a simple hash list, where each transaction hash is simply concatenated, is that it allows efficient membership proof for transactions in the block. In other words, the data structure allows clients to verify that a transaction is part of a given block in time $\log_2(N)$, for a tree with $N$ leaves, as opposed to time proportional to the

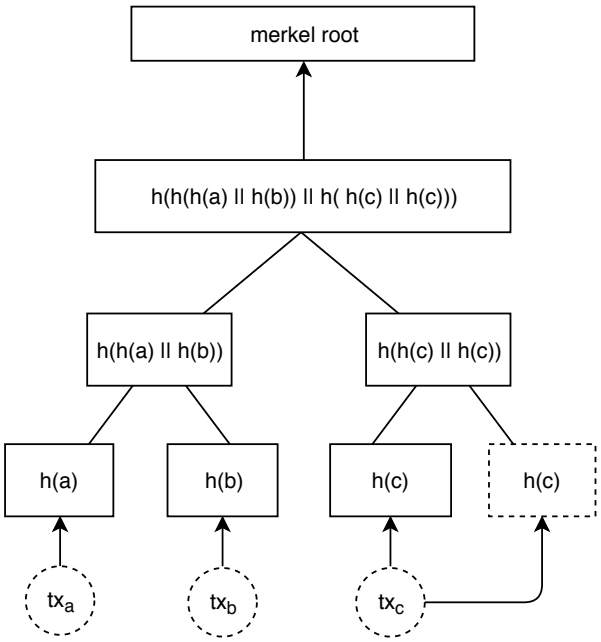


**Figure 5.2:** Merkle Tree Construction. Transaction $tx_c$ is paired with itself, since there are an odd number of nodes in the first row of the tree.

number of leaves as is the case of a hash list [26]. Furthermore, because the block's transactions are only hashed indirectly though the Merkle root hash, the header always has the same length, which means that hashing a block with one transaction takes the same amount of work as hashing a block with thousands [31].

**Timestamp** This header field contains a Unix epoch timestamp, recording the UTC time when the miner began working on generating the block. It is important to note that this timestamp is reported by the miner itself, and that individual timestamps are not reliable since we cannot achieve global, synchronised time in a distributed system (Orlandi [17]).

The block timestamps are primarily used by the difficulty adjustment algorithm to keep the block time at the desired interval. Even though the algorithm takes the inherent imprecision into account, the timestamp still needs to approximate the correct time with reasonable accuracy. To achieve this, the validation engine rejects incoming blocks with timestamps that are either more than 10 minutes in the future, or before or equal to the median timestamp of the previous 11 blocks. Note that the timestamp is restricted according to the median of the previous blocks in contrast to a period of time in the past. This allow nodes to validate any block in the blockchain regardless of the current time and enables nodes to catch up to the rest of the network by processing blocks much later

than they were initially propagated.

**Target** The header contains a target as defined in the following definition:

**Definition 5.2** (Target)**.** The target is a 256-bit unsigned integer threshold which a block's header hash must be equal to or below in order for that block to be valid.[18, 20]

**Definition 5.3** (Difficulty)**.** The difficulty is a measure of how difficult it is to find a hash below a given target. Given the target $t$ of a block $b$ we calculate the difficulty of the block by

$$\text{diff}(b) = \frac{2^{256}}{t}$$

which is the expected/average number of attempts that were necessary to mine the block [14, 47].

From definition 5.2 and 5.3 it is clear that there exists a linear relationship between the target and the difficulty. Indeed, when the target changes by $p$ percent, the difficulty is adjusted by the inverse, $-p$ percent. Because of this close relationship, the two terms are often used interchangeably in the literature.

Blocks are generated on a somewhat regular interval thanks to the difficulty adjustment algorithm discussed in section 7, which actually adjusts the *target*.

**Signature & Public key** The *signature* and *public key* fields of the block's header are used in the mining process and discussed in detail in section 6.

**Transactions list** While the aforementioned fields compose the block's header, the transactions list can be considered the body of the block. The reason for this partition is that the transactions only contribute to the block's hash indirectly through the Merkle root hash as discussed above. There are no constraints on the ordering of transactions in the list, but once it has been chosen by the miner it cannot be altered without invalidating the Merkle root hash. Furthermore, while there is no upper bound on the number of transactions in the list, it is limited by the consensus rules, which state that a block can have a maximum size of 100 KiB.

This limit was chosen based on the Bitcoin block size limit of 1 MB [20], which has been subject to much controversy [30]: To allow for more transaction per second, certain Bitcoin users have been advocating for an increased block size [29]. Of course, the disadvantage of this proposal is that the cost of running a node would increase comparatively, and since the security of the system relies on most users being honest Nakamoto [16], increasing the block size could affect user security [41]. To encourage decentralisation we have chosen the relatively moderate block size limit of 100 KiB in Aucoin.

# 6 Mining Blocks

Since there is no central authority to issue coins into circulation, a way to distribute them is needed. Aucoin follows a similar approach to Bitcoin, in that

coins are introduced through *coinbase transactions* [43]. The miner construct blocks containing a coinbase transaction as the first in the transactions list, awarding new coins to itself.

## 6.1 Coinbase Transactions

As shown in section 4, transaction chains reassign ownership of coins from a previous output. The first transaction in every chain is a coinbase transaction; it does not reassign value, but instead starts a new chain. In Aucoin, the coinbase transaction is a simplified version of a standard transaction; it allows only one input and one output. The value of the output in the coinbase transaction reward the miner of the block 100 Aucoins, as a reward for having expended CPU-time mining it. In addition, the miner must collect the transaction fees of the transactions included in the block. If the value of the output is not equal to the sum of these two amounts, the block is considered invalid.

The single input of the coinbase transaction is a special *coinbase input*. The *public key* and *signature* fields of the standard input are replaced with a *coinbase* and *height* field. The *coinbase* field serves no purpose, but allows miners to include 100 arbitrary bytes in the block. In Bitcoin, the field is often used to trigger a new block hash through the Merkle tree [31]. The *height* field is required to counter an attack involving the construction of multiple transactions with identical hashes.

First, notice that it is difficult for an attacker to control the hash of a regular transaction, since it includes references to previous transactions, possibly outside the attacker's control. Now consider a coinbase transaction; it does not reference any previous outputs in the input, and as such, all fields are either static, or can be chosen freely by the miner. This allows an attacker to mine two different blocks with duplicate coinbase transactions, and by building chains from these, create duplicate standard transactions as well [46]. This possibility can create major problems for the clients, since transaction chains build on the assumption that transaction hashes are unique. In fact, multiple Bitcoins have become unspendable, and thus lost forever, due to this attack [9]. By requiring that the height of the block containing the coinbase transaction is included within, transaction uniqueness is enforced [1].

## 6.2 The Mining Process

Mining blocks is the process of continuously hashing the block header until the hash value is below or equal to that of the block's target. To achieve a different hash at each iteration, a section of the header needs to be changed between attempts. In Bitcoin, this section is a nonce that starts at 0 and is incremented for each attempt [34]. At first, this approach might seem to generate the same sequence of block hashes for every participating miner. It is extremely unlikely, however, for two miners to have the same Merkle root hash because the first transaction in each of their blocks assign the block reward to themselves, ultimately resulting in different hashes. [31]

It is important to note that a miner cannot begin working on a block in advance; this is due to the fact that each block contains the header of the

previous.

### 6.2.1 Mining Pools

Because it is impossible to mine the next block in advance, all miners on the network compete to find a satisfactory block hash first. While not having a significant effect on the expected payout, the variance can be dramatically decreased if a group of miners pool their resources together in a *mining pool*, sharing their processing power and splitting the rewards [35]. Most pool setups include a coordinator who issues blocks with a target that is easier than the real difficulty of the network. The members work on the block, sending solutions back to the coordinator; this allows the members to prove how many resources they spent working on the problem. If a found hash is low enough to also pass the real required difficulty, the block is broadcast to the entire network [35].

The blocks issued by the coordinator includes a coinbase transaction that pays the reward to an address under his control. It is important to note that a miner cannot award the block reward to itself, because doing so would invalidate the solutions sent to the coordinator.

### 6.2.2 Disincentivising Mining Pools

To disincentivise the use of mining pools, and thereby increase the decentralisation of the network, Aucoin draws inspiration from a mining algorithm known as *Sign to Mine*, pioneered by the cryptocurrency ziftrCOIN [15]. The nonce in the block header is replaced with an ECDSA signature and associate public key, resulting in two new block header fields; *signature* and *public key*, as per figure 5.1. These fields are similar to those of the transaction inputs, signing all the header fields except the *signature* itself. Since both the signature and public key contribute to the hash of the block, miners iterate by continuously re-signing the block. Albeit slower than simply incrementing a nonce, the method results in different hashes in each iteration because each signature must use a new secret random number in the signature generation process [23].

Pooled mining is disincentivised by introducing two additional consensus rules; first, the signature must sign the other fields of the header, and secondly, the signature must be produced using the private key associated with the address receiving the block reward [15]. Essentially, to carry out the mining process, the miner must be able to spend the rewards for the mined block. Of course, pooled mining is still possible, but would require all members to possess the private key, allowing them to create transactions transferring the entire block reward to themselves, essentially stealing it. In this sense, pooled mining is still possible, but disincentivised because each member has to trust the entire group.

### 6.2.3 Possible Problems Caused By Sign To Mine

While Sign to Mine initially increases the decentralisation of the network, it reintroduces the problem of large variance in the reward for the miners. This leads to centralisation, because only miners with enough hashing power to get regular payouts will be able to mine, since small constant payouts are much

preferred to the hope of one large reward [5]. Moreover, the system can be worked around by setting up the pool in such way that each participant mines blocks using a private key known only by the pool operator and himself. In this way, it is easy to identify which user stole the coins, and the pool operator could simply require a deposit equal to the block reward, which could be used to cover the potential loss [5]. Of course, some cryptocurrencies may offer block rewards of such high value that very few individuals are able to pay the deposit, leading to even further centralisation.

## 6.3 Most Profitable Transactions

Finding the most profitable set of transactions to include in a block is essential in ensuring the highest possible profit for a miner. Implementing such an algorithm does not increase the security of the protocol, but instead the incentive to mine. Miners store broadcast transactions in a pool in memory (*mempool*). The miner can chose to include transactions from the mempool arbitrarily, or to not add any transactions from the mempool at all, including only the coinbase transaction. Bitcoin clients use heuristics to sort the transactions and pick those with the highest fee/size ratio. In fact, this is an approximation of a NP-hard problem known as knap sacking[6, 44]. We will compare two different algorithms inspired by Bitcoin: The first is a simple algorithm that first excludes any transactions which depends on another transaction in the mempool. It then sorts the remaining transactions based on the $fee/size$ ratio before filling the block until it is full. We shall now more formally define this simple problem.

**Definition 6.1** (Simple most profitable transactions)**.** Given a set of transactions $T$, let $v_i$ be the fee and $w_i$ be the size in bytes of transaction $i$ in $T$. Let $W$ be the maximal size of a block. The most profitable transactions problem is then formally defined as

$$\begin{aligned} maximize & \sum_{i=1}^{|T|} v_i x_i \\ s.t. & \sum_{i=1}^{|T|} w_i x_i < W \text{ and } x_i \in \{0,1\} \end{aligned}$$

The above definition omits the case where transactions are dependent upon each other, which is sometimes a reality in Aucoin. We therefore propose another algorithm that extends the heuristics of the simple algorithm.

### 6.3.1 Approximation in Aucion

The basic idea for solving the most profitable transactions problem is to maintain a list of dependency trees that are constructed in the following way:

- For each transaction $tx$ in the mempool, let $tx$ be the root of a tree.
- Let every transaction that $tx$ depends upon be a child of $tx$.

- Repeat the above steps dynamically until a tree has been constructed for every transaction in the mempool.

The resulting list of trees is sorted by the ratio between the sum of fees and sum of sizes of each tree. The miner can now fill the block using this list by continuously adding all transaction from the most desirable tree. If all transactions of a tree does not fit it is skipped. The process continues until the block is full or we have no more trees to inspect.

The extended algorithm only visits each transaction once when creating the trees, and then sorts the list of trees, yielding a running time of $O(N \log(N))$, where $N$ is the number of transactions in the mempool. Comparing to the simple algorithm, which also has a running time of $O(N \log(N))$, we simply add another layer of heuristics on top. This makes it possible for a miner to include blocks which are dependent on each other, potentially increasing the block reward.

### 6.3.2 Comparing the Algorithms

A simulation generating 1000 transactions was used to compare the two algorithms. Figure 6.1 shows the results of the simulation. Notice that when 0% of the transactions are dependent, the two algorithms yield the same result, but as the number of dependent transactions grow, the extended algorithm outperforms the simple one by having a greater total fee. Another important factor is running time: As shown in figure 6.2, the running time of the extended algorithm suffers greatly as the number of independent transactions grow. The experiment shows that the extended algorithm is approximately a factor of 29 times slower than the simple algorithm with the simple algorithm taking 0.093 seconds to assemble the list of transactions versus the extended algorithm at 2.631 seconds.

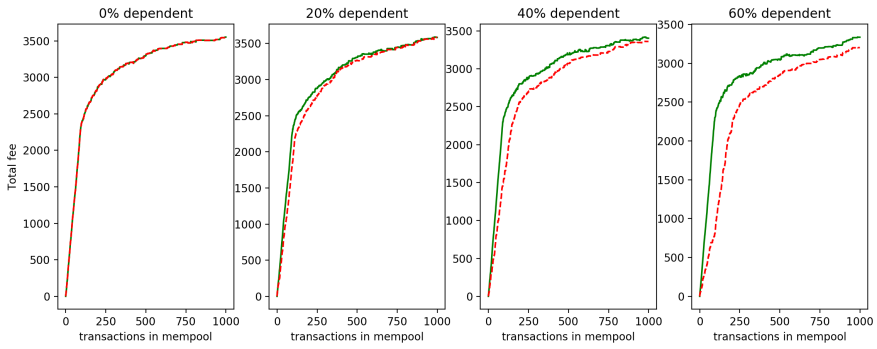


**Figure 6.1:** A simulation that compares the simple (red) and extended (green) algorithm for finding the most profitable set of transactions in the mempool.

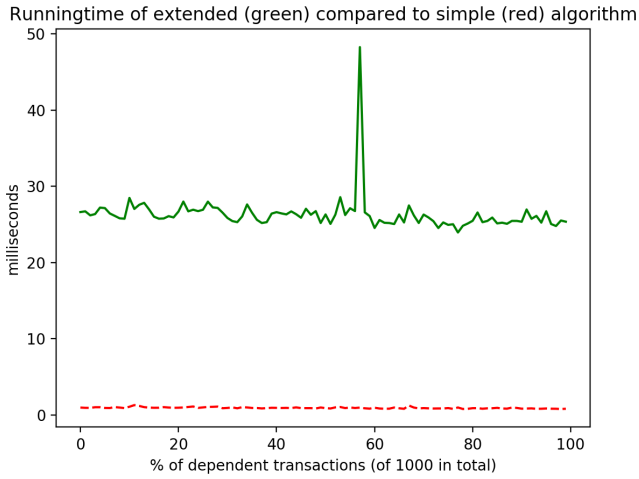


**Figure 6.2:** This graph compares the running time of the simple (red) and extended (green) algorithm.

### 6.3.3 Choosing an Algorithm

The question is then, if one should use the simple or the extended algorithm. In the case of Aucoin, the extended algorithm was chosen, resulting in more CPU-time but better profitability when transactions depend upon each other.

# 7 Difficulty Adjustment

A very important aspect of cryptocurrencies based on PoW is how we adjust the difficulty of mining new blocks. That is, we do not want miners to be using too much or too little time and effort mining blocks. In Aucoin, we strive for a mining time for each block at 60 seconds. In the following section we address the challenges of obtaining this goal, and derive a solution that proves very effective at holding the time span of blocks at the desired interval.

## 7.1 A PID Implementation

Aucoin implements a difficulty adjustment algorithm that adjusts the difficulty for every new block that is found. It draws inspiration from signal processing and regulation by using PID Regulation, which we will discuss now.

**Definition 7.1** (PID Regulation)**.** Given a signal source $s$, a target $o$, and an error function $e(t) = o(t) - s(t)$, we have an algorithm $u$ which regulates the source $s$ such that it limits the error $e(t)$. We define PID Regulation as

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt},$$

where $K_p, K_i, K_d \in \mathbb{R}$ are constants for the proportional part, the integral part and the differential part of the expression, hence the name PID [12].

Aucoin uses the PID Regulation algorithm by defining the error function as

$$e(t) = block\_time - (t_{b_n} - t_{b_{n-1}})$$

where $t_{b_n}$ is a timestamp for block $b_n$, $t_{b_{n-1}}$ is a timestamp for block $b_{n-1}$, and $block\_time$ is the desired time span between mined blocks. Instead of integrating from time 0 till time $t$, Aucoin simplifies this and integrates only over the last 300 blocks. One problem is then to figure out fitting parameters $K_p$, $K_i$, and $K_d$. The consequence of each parameter is discussed by conducting the following experiments. In the first experiment the hash rate is kept constant, and in the second it is variable, but both simulate a network of miners.

### 7.1.1 Previous Work

The use of PID regulation for difficulty adjustment in blockchains is somewhat limited. As shown in the next experiment, PID regulation faces a number of challenges when used with blockchains, which might explain why the method is not widely used. Even though few cryptocurrencies implement algorithms based on PID regulation, developers have shown interest in the idea. For example, a member of the ZCash community conducted experiments on the use of PID regulation in the cryptocurrency ZCash [49]. His results showed that PID regulation algorithms are generally not better than other difficulty adjustment algorithms. However, the experiments were conducted using heavily modified implementations of PID regulation with several signal processing methods applied, which might explain why the results where sub-optimal [49]. The takeaway from these experiments is that needless complexity can cause more harm that good. For this reason, great care have been taken to not repeat these mistakes in the implementation of the Aucoin difficulty adjustment algorithm. We propose an experiment that focuses on a testing a simple implementation of PID regulation using a Monte Carlo simulator.

### 7.1.2 Aucoin PID Regulation Implementation

Because substantial changes in the difficulty between two blocks is undesired, the PID algorithm of Aucoin has a factor of $\frac{1}{10.000}$ applied to the result of the PID calculation to limit the rate of change. Algorithm 1 suggests an implementation of the PID algorithm used in Aucoin.

One problem of the suggested implementation is to figure out parameters $K_p$, $K_i$, and $K_d$. Lets discuss the consequences of each parameter by conducting an experiment: In the experiment we simulate a network of miners, where the hash rate of the network is simulated using a Monte Carlo simulation as described in algorithm 2.

The simulation takes as input a mean value $\mu$ and a standard deviation $\sigma$, and uses these to create a set of $n$ points that are generated using a normal distribution. The series of hash rates is then found by multiplying the last known hash rate with a point from the distributed set. For the actual Monte

**Algorithm 1** PID regulation algorithm used in Aucoin

```
def required_target(block, prev_block, prev_prev_block, k_p, k_d, k_i):
  # Calculate error using the 2 previous blocks
  error, timespan = calculate_error(prev_block)
  last_error, last_timespan = calculate_error(prev_prev_block)

  # Calculate future predictions using slope of error
  derivative = (error - last_error) / timespan

  # Calculate past errors
  integral = sum(get_last_errors(300))

  # Calculate the PID regulation
  pid = (k_p * error + k_i * integral + k_d * derivative)/10000

  # Apply to the previous target
  new_target = prev_target * (1 - pid)
  return new_taret.to_bytes()
```

**Algorithm 2** Monte Carlo simulation of hash rates

```
hash_rates = [100_000]
changes = numpy.random.normal(mean, std, n) + 1
for x in changes:
        hash_rates.append(x * changes[-1])
```

Carlo simulation used to generate a series of hash rates, the hash rate of Bitcoin between May 2017 and May 2018 was used. In this period the mean was $\mu = 10.31 \cdot 10^{-4}\%$ and the standard deviation was $\sigma = 24.64 \cdot 10^{-2}\%$ over a one minute window [4].

### 7.1.3 Experiment

Choose $K_p = 10$, $K_i = 0$, $k_d = 0$. Now run the simulator and consider the results in figure 7.1 on page 23. The observations show that the difficulty is adjusted fast, and the average block time span is very close to the target, even though the hash rate changes dramatically. The geometry of the difficulty has many spikes, indicating that the PID regulator may be overfitting the input signal. The histogram of figure 7.1 also reflects this, as it has a very high probability mass centred around the Poisson distribution. There are 2 major spikes in the histogram, which also indicates overfitting. The calculated mean is 59.59 and the standard deviation is 8.91, which is close to the desired mean of 60. The standard deviation is also around that of the Poisson distribution $\sqrt{60} = 7.746$. This looks good, but there is still room for improvement.

The way to improve the result is to change the parameters of the PID algorithm. It is preferred to have a smoother difficulty adjustment such that we do not adjust the difficulty too much when being victim of noise and stochastic variables. Now, choose $K_p = 2$, $K_d = 0$, $K_i = 0.5$, and consider the results in figure 7.2 on page 24. The results show a mean $\mu = 58.243$ and a standard deviation $\sigma = 11.719$. The problems we encountered in the previous experiment, regarding overfitting and the uneven geometry of the difficulty curve, are now resolved. We now have a very good fit against the poison distribution, and the probability mass is very dense, revealing a great fit. The mean is $\mu = 58.243$, which is a bit worse than figure 7.1, but that is to expected with more generality. The standard deviation $\sigma = 11.719$ is also a bit worse than the first experiment, but the geometry of the histogram in figure 7.2 is more preferable than that of figure 7.1, as it has no significant outliers. By applying the integral part of the PID, we introduce a new problem that is to be seen at around time 30000 in figure 7.2. The difficulty, as well as the average block time span overshoots the desired target before returning to it again. This is a consequence of the integral part, but luckily we can reduce overshooting by using the derivative part of the PID.

Figure 7.3 is the result of running the simulation with parameters $K_p = 2$, $K_d = -50$, $K_i = 0.5$. The main thing to notice here, is that the overshooting has been reduced compared to figure 7.2. The result is visually promising, but once we analyse the statistics of the result we are not so fascinated compared to the experiment of figure 7.2. The mean is $\mu = 58.158$ and the standard deviation is $\sigma = 11.192$, which is a poor improvement considered to figure 7.2.

The explanation for why the derivative part of the PID does not deliver on its promise might be a consequence of the PID algorithm's simplicity. When applying the derivative part, we only look at two unique block timestamps, which are distributed over a Poisson distribution. This makes it absolutely impossible to say that timestamps between blocks are to be considered precise and correct. On the contrary, we might have a very high variance when considering such

timestamps. This could in turn be reduced by implementing the P and D of the PID algorithm in another way; for every block we wish to consider, take the previous $k$ blocks and calculate the mean time span. This could effectively reduce the variance when computing the P and D. We could then discuss whether to call it a PID algorithm or an III algorithm, as it is simply three averages over different windows. For now, we keep the PID regulation algorithm as our difficulty adjustment because it offers sustainable performance and results.

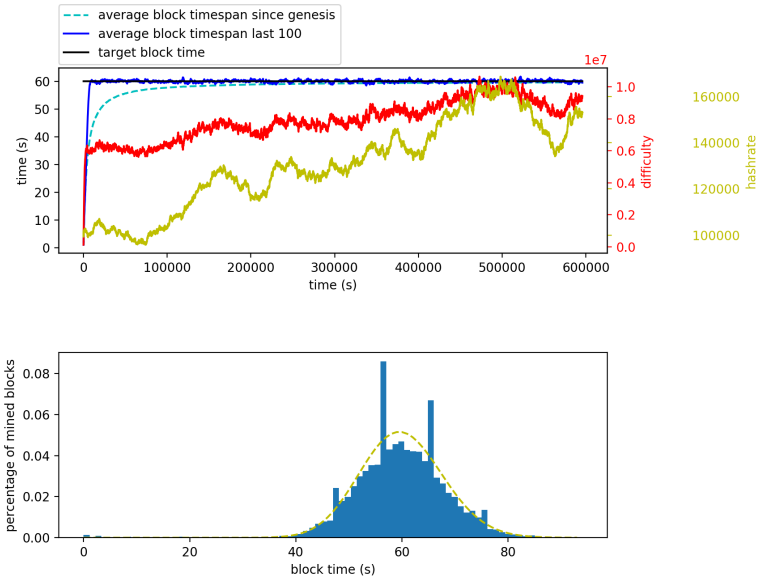


**Figure 7.1:** PID experiment with $K_p = 10, K_i = 0, K_d = 0$.

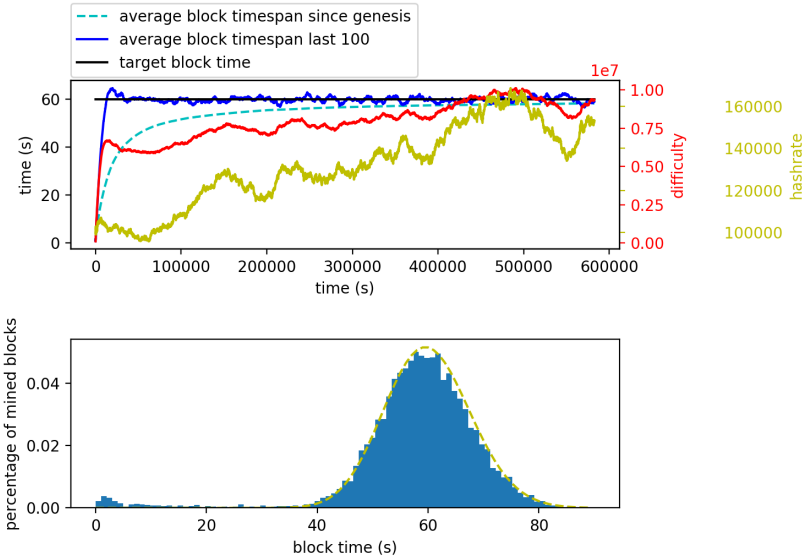


**Figure 7.2:** PID experiment with $K_p = 2, K_i = 0.5, K_d = 0$.

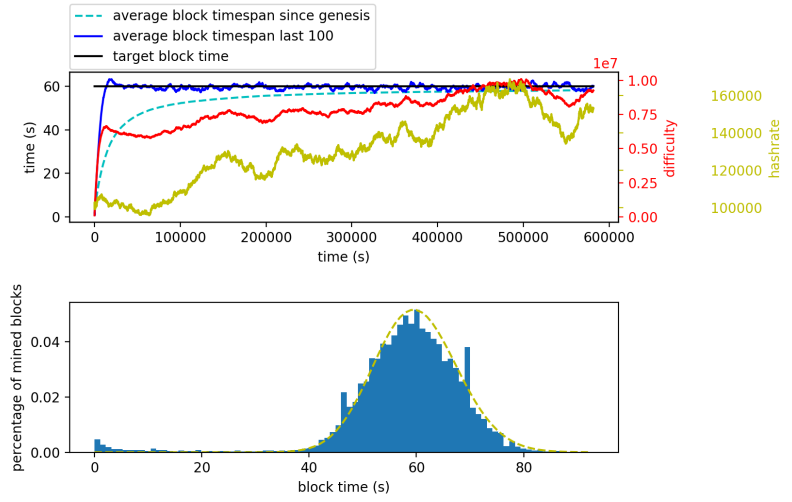


**Figure 7.3:** PID experiment with $K_p = 2, K_i = -50, K_d = 0.5$.

# 8 Network

Clients of bitcoin use a broadcast network to propagate messages to other peers. Such messages will include requests for blocks and transactions, actual block and transaction data, as well as other types of messages [36]. When developing a cryptocurrency, we want it to be distributed and accessible. In the following sections we introduce the theory of gossip networks and pandemic messaging while addressing the challenges of a distributed peer-to-peer network, before returning to the implementation in Aucoin.

## 8.1 Gossip Networks

Implementing a distributed peer-to-peer network requires the designer to consider a number of challenges. If a new cryptocurrency finds traction and needs to scale, it is of uttermost importance that the network does not fail to do so at an application level. The currency will undeniably fail if communication between the peers of the network is not reliable to some extent. For this reason, Aucoin implements a gossip protocol at the application level for multicasting data to peers in the network.

Before continuing, we need to discuss the structure of the network, how peers are connected to each other, and how to ensure that everyone will receive messages from the network. There are generally two approaches; using an overlay network that is structured as a tree or using an overlay network that is structured as a mesh. If using a tree, we guarantee that every peer is connected to the entire network and that there will always be a path from one peer to another peer. This ensures that every peer can communicate with every other peer. Multicast messaging is easily obtained by simply recasting a received message to every connected child or grandparent except the sender. This approach comes at the cost of building and maintaining the tree. For a cryptocurrency, this is not very scalable, especially in scenarios where peers disconnect and connect to the network at will, forcing the tree to be rebuilt needlessly. In addition, while building the tree might seem easy, building an efficient tree may not be as trivial. Because of this trade-off, Aucoin instead uses an overlay network structured as a mesh. With this approach, it is possible for peers to connect and disconnect at any time, without the need of restructuring the overlay network [27]. This network structure will be the basis for the discussion for the remainder of this section.

### 8.1.1 Pandemic Messaging

One of the reasons pandemics are deadly is because they spreads exponentially; When one person gets infected, two persons get infected, four persons get infected, etc. Utilising this property to send messages is precisely what a multicast pandemic protocol does; though, instead of trying to stop the spread it is encouraged. So how do we propagate messages in the network in order to "infect" every node? The solution is to let nodes multicast messages to all its connected peers, or, ideally, a random selection of these. When a node receives

a message it forwards it only if it is the first time seeing it [27]. Consider the following proposition:

**Proposition 8.1.** *Pandemic messaging takes $O(log(N))$ rounds to propagate a single message to all nodes in a network of N nodes.*

*Proof.* Assume that each node in a network of $N$ nodes is connected to $k$ other nodes. Whenever a node $i$ wants to broadcast a message to the network, $i$ can directly message $k$ nodes in one round. Now for the next rounds, every node who received the message can broadcast it to $k$ nodes. Observe that after $n$ rounds, the message must have been broadcast to $k^n$ nodes. So, after $log_k(N)$ rounds, the total number of nodes, who have received the message would be $k^{log_k(N)}$ nodes, thus yielding a bound on the number of rounds it takes to broadcast a message to the network of $O(log(N))$. □

Proposition 8.1 shows that pandemic messaging is fast, but more importantly, scalable. Aucoin uses pandemic messaging at the protocol level for multicasting messages and data across the network. In addition, Aucoin utilises gossiping to limit the amount of traffic the network by only sending blocks and transactions directly, but instead offering them to the network based on the hashes. In this way, receivers can choose to either respond with a request or ignore the offer if they already have received the offered data.

## 8.2 Message types

As discussed, Aucoin has a set of messages which are broadcast around the network. The following exhaustive list of message types is what drives the communication of Aucoin. Every message that is sent through the network has a *message type* field equivalent to one of the types in the list below.

**Hello** The *Hello* message is exchanged between peers who establish a connection. The message has a payload consisting of *version*, *your IP*, and *nonce* fields. The *version* dictates the protocol version of the peer, if the two versions are incompatible the peers will disconnect. The field *your IP* is the IP-address of the other peer, as seen from the perspective of the sender. The field is used to discover a client's own IP-address, even behind firewalls or NATs. The *nonce* field is used to check for self connections, and the method for doing so is described in greater detail in section 8.5.

**Peers** The *Peers* message is sent after receiving a *Peers request* or *Hello* message. The payload of the *Peers* message is a list of IP-addresses known to the sender.

**Peers offer** The *Peers offer* message is sent to offer a *Peers* message to the receiver. A peer will respond with a request only if it does not have the maximum number of connections already.

**Peers request** The *Peers request* message has no payload. It is simply a message letting the receiver know to send known peers to the sender.

**Block** The *Block* message has a block serialised as JSON in the payload. Whenever a *Block* message is received, the receiver broadcasts a *Block offer* message based on the block hash to let everyone in the network know that this block is available in the network.

**Block offer** The *Block offer* message has a block hash in its payload, such that the receiver can search if it has the given block in its blockchain. If not, it may send a *Block request* message back to the sender.

**Block request** The *Block request* message has a block hash in its payload. The receiver of the message will respond with a *Block* message if it has the given block in its blockchain.

**Blocks request** The *Block**s** request* message contains a *header hash* field, which is used by the receiver to find blocks in the blockchain that is between the senders header and what the receiver believes to be the current blockchain header. These blocks are then offered with a *Blocks offer* message.

**Blocks offer** The *Block**s** offer* message has a list of block hashes in the payload. This list is used to request blocks by sending a *Block request* message. The *block hashes* list has a limit, so a *Blocks request* message is sent by the receiver until no block hashes remain in the list.

**Transaction** The *Transaction* message has a transaction object serialised as JSON in its payload. When received, the node broadcasts a *Transaction offer* to the network to let every node in the network know that the transaction is available.

**Transaction offer** The *Transaction offer* message has a *transaction hash* field in its payload. The receiver of this message will searches its blockchain and mempool for the transaction, and if it is not found a *Transaction request* message is returned to the sender.

**Transaction request** The *Transaction request* message has a *transaction hash* field in its payload which the receiver uses to reply with a *Transaction* message.

**Ban** The *Ban* message is sent to blacklist a peer, indicating that the counterpart does not wish to communicate. The message has fields *seconds* and *reason*. The amount of time that the receiver is banned equals the *seconds* field.

## 8.3 Bootstrapping and Discovery

When a client wants to join the network for the first time, it needs a way to discover other peers. The Bitcoin protocol proposes two different bootstrapping techniques; the first way is to receive a list of seed nodes using a number of predefined DNS hostnames. The Bitcoin client will send an *Address* message to these nodes, which will introduce it the rest of the network by forwarding the message to other peers, causing them to establish connections to the new client. If the DNS hostnames fail to resolve the client will fallback to a second bootstrap technique which involves connecting to a number of hard-coded IP-addresses. It goes without saying that this technique is much less robust than the first, since

the list of nodes cannot be updated dynamically without a software update [36]. When the Aucoin client initialises its network it loads in previously known clients and tries to connect to these. If this list is empty, or every connection attempt unsuccessful, the client will perform a DNS lookup to the hostname `seed.aucoin.network` for nodes to connect to. This makes the DNS lookup the last resort, as we always try connecting to previously seen nodes first.

Whenever a client establishes a connection to another node, it receives a list of potential peers to connect to. Even if a node already has the maximum number of desired connections, it will always send this list to connecting peers to ensure that the counterpart has someone to connect to. This list will be added to the client's set of *known peers*, and will in turn be broadcast along to other nodes in the network. If an IP address from the set is unreachable, it will be removed.

## 8.4 Rejecting and Banning Connections

There are several reasons for a connection to another node to be rejected: A peer can be banned if the first message it sends is not a *Hello* message, if it sends malformed data, or if the requested part was forced to reject the connection due to already maintaining the maximum number of desired connections. By default, a peer is banned for 60 seconds, and if it is following the protocol it will not try to connect during this interval. Aucoin restricts a peer from connecting to the same peer twice, as well as restricting connections to itself. Self connections are detected by comparing the IP-address of the connection with its own address as reported by the other peers. Additionally, a peer may detect a self connection by comparing the nonce of the *Hello* message to its own 32-bit nonce. These two techniques are discussed next.

## 8.5 Distinguishing Peers

It is important to be able to detect self connections since we do not want a peer to unnecessarily broadcast messages to itself. Furthermore, doing so might mess with the internal events on an application level. Internal events like block discovery and newly added transactions could interfere with each other if they are being fired both internally and through the network layer, which might introduce behaviour which is hard to debug. In this section, we propose two different solutions to find if a peer is connecting to itself. These methods work in unison in Aucoin, and are inspired by the implementation of the network layer of Bitcoin.

### 8.5.1 IP-address Discovery

The implementation of the original Bitcoin client used public web services to retrieve its external routable IP-address. The client would connect to two hard-coded services; `checkip.dyndns.org` and `www.showmyip.com`. If either service returned an IP-address, the node would use the result when broadcasting its presence to the network [40]. This is not truly a distributed solution, and might introduce security risk if the operator of any of these services were to return

malicious results. Aucoin, instead, seeks to solve the problem of discovering ones own IP-address in a distributed manner.

The solution we propose works by letting each node send the perceived address of the other peer through the *your ip* field on the *Hello* message. Each node then collects the responses in a map with the sender's address as the key. In this way, each peer may only "vote" on one address. To find a node's own IP-address, it is sufficient to follow a simple heuristic; let $n$ be the number of IPs in the map. Now find the most common IP-address among the values. Denote the number of occurrences $k$. The own IP of the node can then be determined with confidence $\frac{k}{n}$, under the assumption that the majority of peers in the network are honest.

### 8.5.2 Nonce Control

Another, perhaps more robust method of detecting connections to oneself, is to generate a nonce when starting the client. Choose this nonce randomly with a length of 32 bits. Then, when establishing a connection to other peers with the *Hello* message, simply send this nonce in the payload. Now the receiver can check if the nonce matches its own, and if so, rejects the connection because it. There is one problem to consider with this approach though. We choose 32 bits randomly under the assumption that this nonce will be unique on the network. For a large network, this assumption may not hold, since it may be the case that multiple peers select the same random 32 bits.

For a network of $k$ peers, the probability of two peers having the same 32 bits is $P(x = y) = \frac{k}{2^{32}}$. Furthermore, this does not seem to pose a problem for massive cryptocurrencies like Bitcoin, which currently has around 10,000 peers [3]. By using these numbers, the probability of randomly selecting the same nonce for nodes in a network of such size is $\frac{10000}{2^{32}} = 2.3 \cdot 10^{-6}$, which surely is very low.

### 8.5.3 Network Sniffing

A concern is that an adversary can sniff the data exchanged between peers `in a Hello` message. One can argue that no adversary can use this data in an attack which serves to shut out a peer from the network. The reason being that the protocol is independent of trust between peers individual peers. As such, the worst an adversary can do is act as someone else by using that peer's nonce, but this will not harm the network. An adversary can lie when sending the `Hello` message, and give a wrong *`your IP`* to a peer. And, if for some reason, the network consists of more than 50% adversaries who agreed on the same wrong IP address to send to this peer, it is possible that the peer could be triggered into thinking that this IP-address belongs to it. But, again, this would in no case be fatal to the network nor peer, as the peer would still identify when connecting to itself by other measures as discussed in above.

## 8.6 Max Peers

A very important piece of heuristics in the Aucoin client is the `max-peers` variable. This variable controls how many peers a client can initiate communication with.

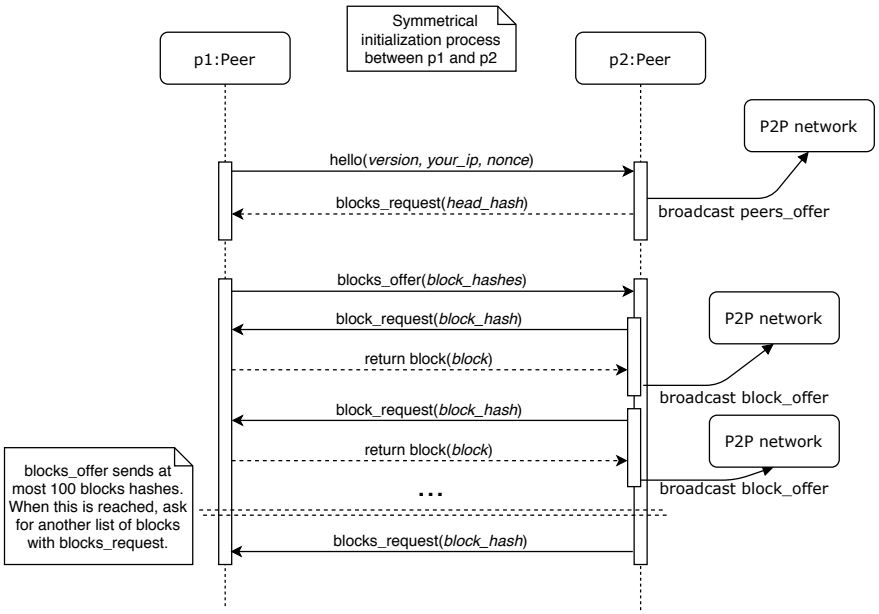


**Figure 8.1:** A sequence diagram of the establishment of a connection between two parties. The reader should view the sequence diagram as a symmetrical process for both p1 and p2. This means, that while p1 is performing this sequence, p2 is performing the symmetrical sequence asynchronously.

The default is 4, which, in an experimental setting with small networks, is fine. In a real scenario, this could be higher. One thing to notice, is that an Aucoin client can have *more* connections than the `max-peers` variable implies. This is due to heuristics in the network layer. A client should never be selfish and connect to only the peers he wants to talk to. If this was the case, then 5 peers could connect to each other and form a densely connected network that no one else could join. Therefore, the limit of *actual* connection that can be made to a client is `max-peers` $\cdot 3$, such that there is always twice as many seats for peers wanting to join the network.

## 8.7 Initial Block Download

When a peer has established a connection to the network it will receive a *Hello* message, as described above. It will then send a *Blocks request* message, with its blockchain block header in the payload. This is done to find if the peer is any blocks behind the current consensus. If it is, it will receive these blocks from *Block* messages, and iteratively add these blocks to its blockchain. Every block is validated during this process to ensure a valid blockchain.

A sequence diagram of the establishment of a connection between to parties is shown in figure 8.1.

## 8.8 Experiments

To verify the implemented network architecture, extensive experimentation was made. An important benchmark is the the latency and connectivity of the network. The first experiment we will discuss, serves as argument for why the network will scale and keep a low latency.

Examine figure 8.2, which is the result of the first experiment where the latency of transaction transport in networks of different peer sizes was recorded. This was truly a challenge to implement and to automate. In each iteration, that is for each network of a given size, a seed node was started and then a number of non-mining clients was also started. The `max-peers` was chosen to be 4, which means that each peer in the network could obtain up to 12 total connections. Whenever a client received a new transaction and it was validated, a `HTTP` post message was sent to a specifically designed test server, to be able to collect a synchronised timestamp of when transactions was received from peers. Each client was assigned an artificial network latency of 300ms to simulate a real internet connection. Figure 8.2 show that there is a somewhat sub-linear relation between the time it takes to send a transaction, and the number of peers in the network. This is a great discovery that reveals the truth of proposition 8.1. The median time reflect the proposition well, whereas the total time it took a transaction to be multicast in the network begin to show strange behaviour in networks of sizes above 80. The explanation for this behaviour could be that the experiment was performed on a single computer where strange network behaviour began when the CPU was overloaded and when the number of ports in use was high.

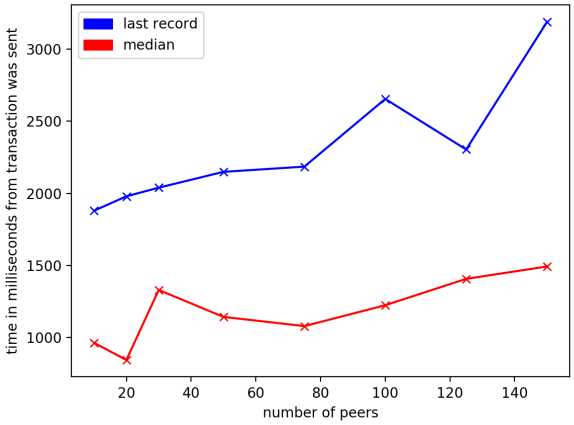


**Figure 8.2:** The diagram illustrates the time it took an experiment to broadcast a transaction to all peers on the network.

The second experiment serve show that the choice of max-peers directly affects the connectivity of the network. The three first diagrams of figure 8.3 have `max-peers` set to 4, and the last three diagrams have `max-peers` set to 8. One thing to notice, is that the seed is not necessarily the node with the most

connections. In almost all diagrams the seed is not dominant, indicating that the seed is of less importance once the network is established. Even with a small `max-peers` setting, the seed will help new nodes be discovered on the network.

The second observations of the experiment is that the higher `max-peers` is, the more connections a network has. This is obvious, but it should be noted that more connections result in a highly coupled network. This might be good, as messages then pass through less layers of peers, and thus multicast faster, but it might also be an overhead for network traffic. On one computer, higher `max-peers` means unresponsive CPU, due to many connections, so we decided for a small `max-peers` during experimentation and test of the Aucoin client. In a real scenario, where you do not run 50 clients on a single machine, this conclusion could be different.

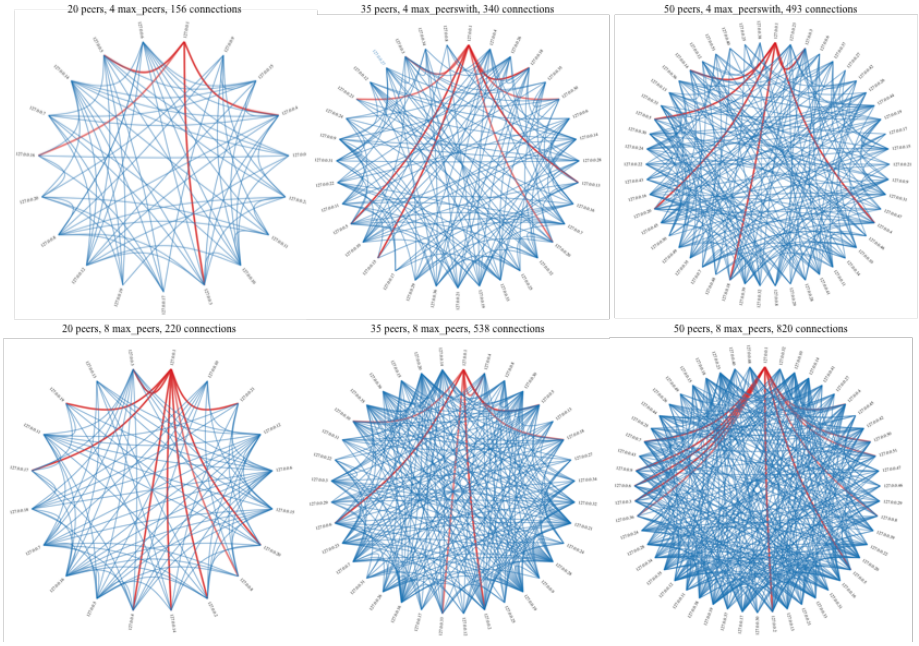


**Figure 8.3:** The diagrams show the connectivity of networks of different sizes and with different `max-peers` variable.

# 9 Data Storage

In this section, we consider the implementation of the data structures of the Aucoin blockchain. To begin, let us discuss previous work by exploring the Bitcoin core client: Bitcoin uses LevelDB, a key-value store, as the database engine for storing blocks and UTXOs. Blocks are dumped on disk when received from the network, and metadata describing each block is stored in the LevelDB database. Yet another LevelDB database is used to maintain the UTXO set, such that it can be easily searched through when validating blocks and transactions. The point of having a separate database for UTXOs is to decrease the time spent looking up transaction outputs. If no database for the UTXOs where present, one would have to search the entire database of blocks, visiting their data on disk, to validate incoming blocks and transactions [48]. As of today, the size of the bitcoin blockchain is about 150GB, making it obvious that maintaining a smaller database of UTXOs is preferable [25].

## 9.1 Data Storage in Aucoin

One drawback of the way Bitcoin handles its blockchain data structure is that it is slow to rebuild the UTXO set if it is lost. Hence, a new peer has to revalidate and lookup the blocks on disk through the LevelDB database [48]. This is a bottleneck that we wish to address and improve upon in Aucoin.

Instead of having multiple key-value databases, Aucoin has a single SQLite RDBMS, which holds the blocks, transactions, and transaction outputs and inputs. For a relational schema of the database design see figure 9.1. An important observation to make when studying the relational schema is that blocks are not stored directly on disk. Instead, they are stored in a relationship model between blocks, transactions, and transaction inputs and outputs. Since every transaction input and output is directly accessible through a single call to the database, it is fast to search for a specific UTXO, transaction, block, etc. Furthermore, being able to express calls to the data storage efficiently using SQL helped make the PID regulation algorithm discussed earlier more stable and faster; while early simulations of the PID algorithm took tenths of minutes to run, using SQL queries for finding means and searching blocks by block heights, we eventually could run the simulations in just minutes.

## 9.2 Adding Blocks to the Blockchain

After a block has passed the preliminary validation checks it is added to the block chain as a side branch off the main branch. It will then have to be determined if the blocks should become the header of a new main branch. Consider the following definition:

**Definition 9.1** (Total work)**.** Given a chain of blocks $C$, we can calculate the total work done to arrive at the chain, by

$$\sum_{b \in C} \text{diff}(b).$$

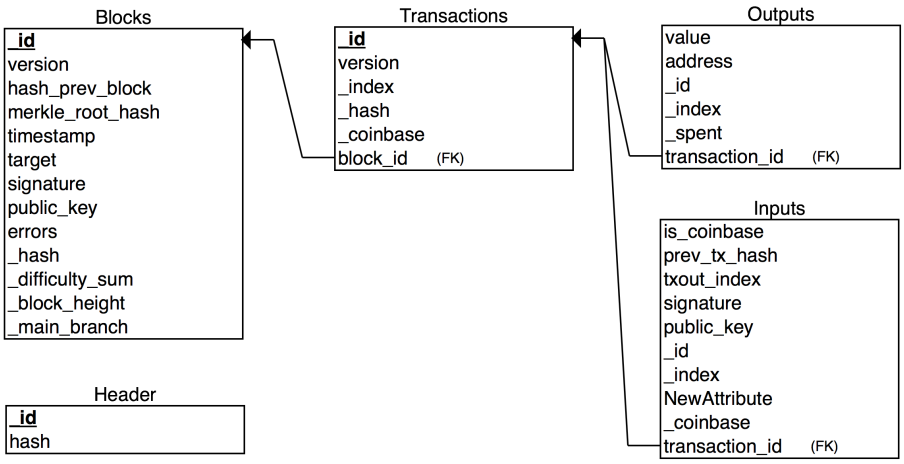


**Figure 9.1:** Relational schema of the SQLite database used in Aucoin. Notice that blocks are *not* directly stored in the format they are received from the network. This creates a responsibility for the client to be able to interpret and encode a block from the database to the data structure required by the Aucoin protocol. However, this is a feature by design.

Total work is the metric for determining if a branch should be considered as the main branch, by which the chain with the most work is the one regarded as valid. The first Bitcoin implementation determined the main branch as the *longest* chain of blocks. This allowed the blockchain to be attacked; consider an adversary mining a branch in private. As it is okay, to solve a block from the past, an adversary could choose to build upon a block several months old. By being dishonest about the timestamps, the adversary could cause the difficulty adjustment algorithm to make block target unnaturally low, allowing him to mine a chain of length much greater than the public one. Note that the total work of this branch would be lower than the public one, since the adversary would have to control more than half the mining power to catch up to the public chain. Using total work as the metric protects from this attack. Algorithm 3 is invoked after a block is added to the blockchain to reorganise the main branch. The subroutine `get_header()` gets the old header of the blockchain, while the subroutine `get_block_with_most_work()` gets the block that is intended to be the new header. Figure 9.2 illustrates an example from running the algorithm for reorganising the main branch after adding a block with total work of 40.

## 9.3 Wallet

The wallet is a data structure containing the private keys of the user. We have seen that it is recommended to often generate new addresses, and the wallet facilitates this. The wallet of Aucoin is directly accessible via the CLI and is encryptable. This grants the user the possibility of securely storing the private keys of the wallet. The technical implementation of the wallet itself is not worth discussing. The part that is interesting, is why it was chosen to use elliptic curves for the signature scheme, as opposed to RSA.

**Algorithm 3** Pseudo code for adding a block to the blockchain

```
reorganize_main_branch():
        header_block = get_header()
        most_work_block = get_block_with_most_work()

        if header is most_work_block:
                return

        fork_block = find_fork()
        while header_block != fork_block:
                remove_from_main_branch(header_block)
                header_block = get_header()

        block = most_work_block
        blocks = []
        while block != fork_block:
                blocks.append(block)
                block = block.previous_block

        for block in blocks.reverse():
                add_to_main_branch(block)
```

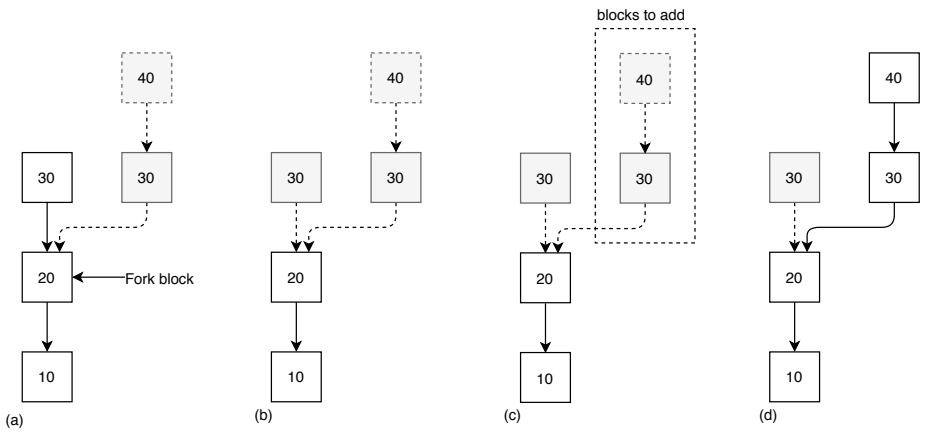


**Figure 9.2:** Example of the algorithm to find the branch with most work. The sum of total work at each block is written inside the blocks. A white block is on the main branch, and a grey is on a side branch. (a) The current header block and max_total_work_block is not equal, therefore a blockchain reorganisation is needed. The algorithm finds the block from which the max_total_work_block forks of off the main branch, as indicated by the arrow. (b) Disconnect blocks from the main branch until the fork block is reached, starting at the header block. (c) Find the blocks to add to the main branch by traversing from the max_total_work_block down to the fork block. (d) Add the blocks starting with the block above the fork block.

RSA is a public key cryptography that builds on the assumption that factoring prime numbers is hard. On the other hand, ECDSA is a public key cryptography that builds on elliptic curves. One of the important features that a crypto-system should have, is that it is a good trapdoor function, which means that it is easy to calculate one direction, but hard to undo. It turns out that RSA is not really a good trapdoor function. Conversely, elliptic curves provide an excellent trapdoor function [7]. Another neat aspect of ECDSA in the application of a cryptocurrency, is that for a 256-bit key, it offers the same amount of security as a 3248-bit asymmetric RSA key. An experiment conducted by Cloudflare [8] shows that it is possible to sign 9516 times per second using a 256-bit ECDSA key versus 1001 signatures per second using a 2048-bit RSA key. This shows that not only is ECDSA faster, it also provides significantly better protection for the same amount of CPU-cycles. Most importantly, the smaller keys of ECDSA means that transactions can be smaller, allow more to be included in a block. The National Institute of Standards and Technology recommends a key size of 256-bit when using elliptic curve cryptography [24], which is why Aucoin uses ECDSA with 256-bit private keys. Compared to RSA, the keys are both more protected, smaller in size, and faster to create signatures with.

# 10 Conclusion

With the introduction of Aucoin, we have created a scalable and reliable cryptocurrency that features a simpler block and transaction structure than that of Bitcoin. This structure delivers protection against transaction malleability, which is still a known weakness of Bitcoin. By implementing Sign to Mine, use of simple mining pools is disincentivised, alternating Aucoin from Bitcoin. Miners of the Aucoin network can profit greater from dependent transactions by selecting transactions in the mempool in time $O(N \cdot \log(N))$. Further, by taking advantage of the possibilities offered by a relational database, nodes joining the network will not have to rebuild the UTXO set, which saves an tremendous amount of time. Aucoin implements a gossip network protocol for multicasting messages in an overlay network structured as a mesh. This allow messages to reach all peers of the network in time $O(log(N))$ with significantly less overhead than a densely connected network. Importantly, seed nodes – arguably the most centralised part of the design – have been shown to be of less importance to the network, which maintains reliable delivery of messages even at low level of MAX-PEERS. The few centralised web-services of Bitcoin have been replaced using heuristics to accomplish decentralised IP-address discovery while still maintaining the security of the system. Lastly, the implementation of a PID algorithm for adjusting the difficulty in the Aucoin network resulted in a stable and almost constant average block time of 60 seconds. While Bitcoin adjusts the block difficulty every 2016th block, Aucoin does this for every new block in the system. This makes Aucoin more resistant by having a smooth difficulty adjustment.

# References


[1] Gavin Andresen. *BIP 34*. 2012. URL: https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki (visited on 15/06/2018).

[2] Daniel J. Bernstein. *Introduction to post-quantum cryptography*. 2009. URL: http://www.pqcrypto.org/www.springer.com/cda/content/document/cda_downloaddocument/9783540887010-c1.pdf (visited on 14/06/2018).

[3] bitnodes.earn.com. *GLOBAL BITCOIN NODES DISTRIBUTION*. 2018. URL: https://bitnodes.earn.com (visited on 06/06/2018).

[4] Blockchain.info. *Hash Rate (Bitcoin)*. 2018. URL: https://blockchain.info/charts/hash-rate?timespan=1year (visited on 04/05/2018).

[5] ZIFTR BLOG. *Goodbye Sign to Mine. Hello Proof of Knowledge.* 2015. URL: https://web.archive.org/web/20160318054440/http://blog.ziftr.com/2015/02/25/goodbye-sign-to-mine-hello-proof-of-knowledge/ (visited on 15/06/2018).

[6] Joseph Bonneau. *Bitcoin mining is NP-hard*. 2014. URL: https://freedom-to-tinker.com/2014/10/27/bitcoin-mining-is-np-hard/ (visited on 04/06/2018).

[7] Cloudflare. *A (Relatively Easy To Understand) Primer on Elliptic Curve Cryptography*. 2018. URL: https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/ (visited on 06/06/2018).

[8] Cloudflare. *ECDSA: The digital signature algorithm of a better internet*. 2018. URL: https://blog.cloudflare.com/ecdsa-the-digital-signature-algorithm-of-a-better-internet/ (visited on 06/06/2018).

[9] Matt Corallo. *duplicate coinbase transactions are allowed, and there are 2 pairs of dups*. 2011. URL: https://github.com/bitcoin/bitcoin/issues/612 (visited on 15/06/2018).

[10] Christian Decker and Roger Wattenhofer. "Bitcoin Transaction Malleability and MtGox". In: *Computer Security - ESORICS 2014*. Ed. by Mirosław Kutyłowski and Jaideep Vaidya. Cham: Springer International Publishing, 2014, pp. 313–326. ISBN: 978-3-319-11212-1.

[11] Manav Gupta. *Blockchain for dummies IBM limited edition*. John Wiley & Sons, Inc., 2017. ISBN: 978-1-119-37139-7.

[12] Tore Hagglund. *PID Controllers: Theory, Design, and Tuning*. ISA: The Instrumentation, Systems, and Automation Society, 1995. ISBN: 1-55617-516-7.

[13] Marius Hanne. *Bitcoin - Stats*. 13th June 2018. URL: https://webbtc.com/stats (visited on 13/06/2018).

[14] Michael Marquardt. *Strongest vs Longest chain and orphaned blocks*. 2014. URL: https://bitcoin.stackexchange.com/a/29744 (visited on 04/05/2018).

[15] Stephen Morse. *What would be required to disincentivize mining pools?* 2015. URL: https://bitcoin.stackexchange.com/questions/26961/what-would-be-required-to-disincentivize-mining-pools/34285#34285 (visited on 15/06/2018).

[16] Satoshi Nakamoto. *Bitcoin - A Peer-to-Peer Electronic Cash System*. 2008. URL: https://bitcoin.org/bitcoin.pdf (visited on 04/05/2018).

[17] Claudio Orlandi. *Synchronization (1) DISTRIBUTED SYSTEMS*. 2017. URL: https://blackboard.au.dk/bbcswebdav/pid-821492-dt-content-rid-1959839_1/courses/BB-Cou-UUVA-64586/41%20Synchronization%281%29.pdf (visited on 23/05/2018).

[18] Bitcoin Project. *Bitcoin Developer Glossary*. 2018. URL: https://bitcoin.org/en/developer-glossary (visited on 23/05/2018).

[19] Bitcoin Project. *Bitcoin Developer Guide*. 2018. URL: https://bitcoin.org/en/developer-guide (visited on 22/05/2018).

[20] Bitcoin Project. *Bitcoin Developer Reference*. 2018. URL: https://bitcoin.org/en/developer-reference (visited on 22/05/2018).

[21] Martin Roetteler et al. *Quantum Resource Estimates for Computing Elliptic Curve Discrete Logarithms*. Cryptology ePrint Archive, Report 2017/598. 2017. URL: https://eprint.iacr.org/2017/598 (visited on 04/06/2018).

[22] David Schwartz. *How does change work in a bitcoin transaction?* 2011. URL: https://bitcoin.stackexchange.com/questions/736/how-does-change-work-in-a-bitcoin-transaction/738#738 (visited on 04/06/2018).

[23] National Institute of Standards and Technology. *Digital Signature Standard (DSS)*. 2013. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf (visited on 15/06/2018).

[24] National Institute of Standards and Technology. *Recommendation for Key Management*. 2018. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57Pt3r1.pdf (visited on 06/06/2018).

[25] Statista.com. *Size of the Bitcoin blockchain from 2010 to 2017*. 2018. URL: https://www.statista.com/statistics/647523/worldwide-bitcoin-blockchain-size/ (visited on 05/06/2018).

[26] Michael Szydlo. "Merkle Tree Traversal in Log Space and Time". In: *Advances in Cryptology - EUROCRYPT 2004*. Ed. by Christian Cachin and Jan L. Camenisch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 541–554. ISBN: 978-3-540-24676-3.

[27] A.S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms. Second Edition*. Prentice Hall, 2007. ISBN: 0132392275.

[28] Billy Tetrud. *Why does Bitcoin send the "change" to a different address?* 2013. URL: https://bitcoin.stackexchange.com/questions/1629/why-does-bitcoin-send-the-change-to-a-different-address/18331#18331 (visited on 04/06/2018).

[29] WeUseCoins. *Why the blocksize limit keeps Bitcoin free and decentralized*. 2013. URL: https://www.weusecoins.com/why-blocksize-limit-keeps-bitcoin-free-decentralized (visited on 14/06/2018).

[30] Bitcoin WIki. *Block size limit controversy*. 2018. URL: https://en.bitcoin.it/wiki/Block_size_limit_controversy (visited on 14/06/2018).

[31] Bitcoin Wiki. *Block hashing algorithm*. 2015. URL: https://en.bitcoin.it/wiki/Block_hashing_algorithm (visited on 31/05/2018).

[32] Bitcoin Wiki. *Change*. 2017. URL: https://en.bitcoin.it/wiki/Change (visited on 04/06/2018).

[33] Bitcoin Wiki. *Contracts*. 2017. URL: https://en.bitcoin.it/wiki/Contracts (visited on 04/06/2018).

[34] Bitcoin Wiki. *Hashcash*. 2017. URL: https://en.bitcoin.it/wiki/Hashcash (visited on 14/06/2018).

[35] Bitcoin Wiki. *Mining pool reward FAQ*. 2011. URL: https://en.bitcoin.it/wiki/Mining_pool_reward_FAQ (visited on 15/06/2018).

[36] Bitcoin Wiki. *Network*. 2018. URL: https://en.bitcoin.it/wiki/Network (visited on 22/05/2018).

[37] Bitcoin Wiki. *OP-CHECKSIG*. 2018. URL: https://en.bitcoin.it/wiki/OP_CHECKSIG (visited on 13/06/2018).

[38] Bitcoin Wiki. *Proof of work*. 2016. URL: https://en.bitcoin.it/wiki/Proof_of_work (visited on 23/05/2018).

[39] Bitcoin Wiki. *Protocol documentation*. 2018. URL: https://en.bitcoin.it/wiki/Protocol_documentation (visited on 22/05/2018).

[40] Bitcoin Wiki. *Satoshi Client Node Discovery*. 2018. URL: https://en.bitcoin.it/wiki/Satoshi_Client_Node_Discovery (visited on 05/05/2018).

[41] Bitcoin Wiki. *Scalability FAQ*. 2017. URL: https://en.bitcoin.it/wiki/Scalability_FAQ (visited on 14/06/2018).

[42] Bitcoin Wiki. *Script*. 2018. URL: https://en.bitcoin.it/wiki/Script (visited on 04/06/2018).

[43] Bitcoin Wiki. *Transaction*. 2018. URL: https://en.bitcoin.it/wiki/Transaction (visited on 04/06/2018).

[44] Bitcoin Wiki. *Transaction fees*. 2018. URL: https://en.bitcoin.it/wiki/Transaction_fees (visited on 05/05/2018).

[45] Bitcoin Wiki. *Transaction malleability*. 2017. URL: https://en.bitcoin.it/wiki/Transaction_Malleability (visited on 13/06/2018).

[46] Pieter Wuille. *BIP 30*. 2012. URL: https://github.com/bitcoin/bips/blob/master/bip-0030.mediawiki (visited on 15/06/2018).

[47] Pieter Wuille. *How does a client decide which is the longest block chain if there is a fork?* 2011. URL: https://bitcoin.stackexchange.com/a/939 (visited on 04/05/2018).

[48] Pieter Wuille. *What is the databases for?* 2018. URL: https://bitcoin.stackexchange.com/questions/11104/what-is-the-database-for/11108#11108 (visited on 05/06/2018).

[49] zawy12. *PID controller difficulty algorithm*. 2018. URL: https://github.com/zawy12/difficulty-algorithms/issues/20 (visited on 22/05/2018).

# 11 Appendix

The source code of Aucoin is available at https://cs.au.dk/~caspervk/aucoin.tar.gz.