UPTEC F 16043
Examensarbete 30 hp
Juli 2016

UPPSALA
UNIVERSITET

A technical overview of distributed ledger technologies in the Nordic capital market.

Ludvig Backlund

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

A technical overview of distributed ledger technologies in the Nordic capital market.

Ludvig Backlund

This thesis examines how Distributed Ledger Technologies (DLTs) could be utilized in capital markets in general and in the Nordic capital market in particular. DLTs were introduced with the so called cryptocurrency Bitcoin in 2009 and has in the last few years been of interest to various financial institutions as a means to streamline financial processes. By combining computer scientific concepts such as public-key cryptography and consensus algorithms DLTs makes it possible to keep shared databases with limited trust among the participators and without the use of a trusted third party. In this thesis various actors on the Nordic capital market were interviewed and their stance on DLTs were summarized. In addition to this a Proof of Concept of a permissioned DLT application for ownership registration of securities was constructed. It was found that all the interviewees were generally optimistic about DLTs potential to increase the efficiency of capital markets. The technology needs to be adopted to handle the capital markets demand for privacy and large transaction volumes, but there is a general agreement among the interviewees that these issues will be solved. The biggest challenge for an adoption of DLTs seem to lie in that of finding a common industry-wide standard.

Handledare: Gustav Ekeblad
Ämnesgranskare: Tjark Weber
Examinator: Thomas Nyberg
ISSN: 1401-5757, UPTEC F 16043

1 Popular scientific summary in Swedish

Blockkedjeteknologin är i grunden en databasteknik som med hjälp av kryptografi och konsensustekniker möjliggör för säkra distribuerade databaser utan inblandning av en tredje part. Teknologin introducerades 2009 som den underliggande tekniken bakom kryptovalutan Bitcoin. Sedan dess har applikationsområdena vidgats och under de senaste åren har diverse finansiella institutioner intresserat sig för vad teknologin kan göra för att effektivisera den finansiella infrastrukturen.

På dagens kapitalmarknader samverkar en mängd olika aktörer såsom banker, börser och värdepapperscentraler. En värdetransaktion involverar flera separata processer hos de olika aktörerna och kan inom EU ta upp till två arbetsdagar att genomföra till fullo. Blockkedjeteknologin möjliggör en högre grad av integration av de olika aktörernas dataregister, något som i längden kan leda till en mer effektiv kapitalmarknad där bland annat motpartsrisker och transaktionskostnader kan minskas.

I det här examensarbetet har en undersökning genomförts för att besvara frågan om vilka tekniska möjligheter och svårigheter som finns med en eventuell implementering av blockkedjeteknologin på den nordiska kapitalmarknaden. Undersökningen har genmförts dels via litteraturestudier och dels via intervjuer av aktörer på den nordiska kapitalmarknaden med insyn i den nuvarande utvecklingen av blockkedjdeteknologin. Utöver detta har även en applikation för att registrera och överföra värdepapper med hjälp av blockkedjdeteknologi konstruerats.

Den övergripande synen på blockkedjeteknologins potential att effektivisera den nordiska kapitalmarknaden är positiv. Teknologin måste dock anpassas för att kunna hantera de stora transaktionsvolymer samt de krav på sekretess som finns inom dagens kapitalmarknader. En övergång från de publika blockkedjorna som idag är dominerande till privata blockkedjor samt mer effektiva konsensustekniker har dock stor potential att lösa dessa tekniska problem. Den stora utmaningen för att till fullo kunna utnyttja teknologins potential ligger sannolikt i att finna en gemensam industristandard.

Contents

2 Introduction

The capital markets consist of a wide variety of actors that have an intrinsic need to interact and share data among one another. The technical infrastructure for doing this is to a large extent unintegrated, and non-shared proprietary databases are generally used to keep record of asset ownership. In the report *Distributed ledger technologies in securities post-trading* released by the European Central Bank in 2016 it is commented on these isolated systems as contributing to an increased operational risk as well as limiting the potential for risk-sharing among European investors due to higher costs of international securities transactions [1]. Distributed Ledger Technologies (DLTs) offer a method to streamline the actions of processing and sharing data between financial actors and have the last few years been the target of intensive attention, both from private companies and governmental agencies [2][3][4]. In figure 1 it is illustrated how DLTs could be used by a consortium of banks to settle trades directly with each other without the involvement of an intermediary. Instead of querying a central depository each bank keeps their own local copy of the asset data, synchronization of the data is guaranteed using DLTs. This way trades and data sharing can be done bilaterally, i.e. directly between two parties.

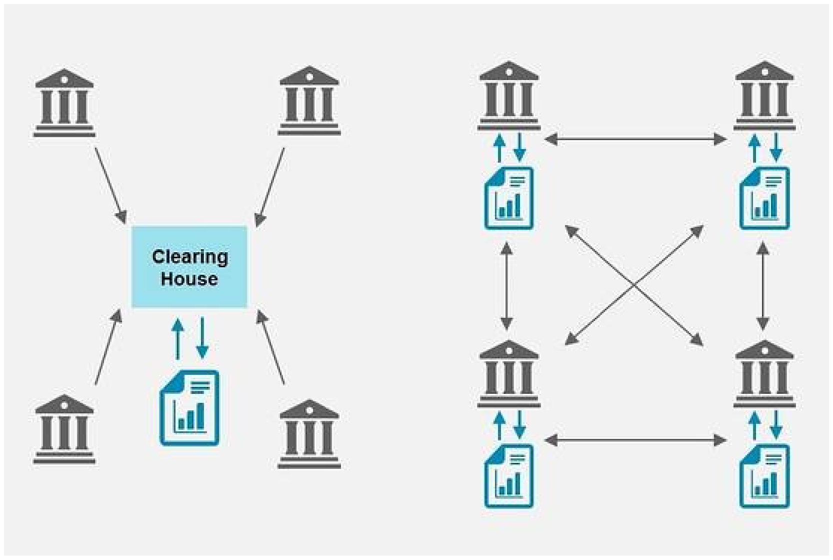

Figure 1: To the left: The current system where a central authority (the clearing house) keeps custody of the asset data. To the right: Each bank have their own copy of the asset data and keeps synchronized using DLTs. *source: wsj.com*

The concept of DLTs was first introduced in 2009 by (the pseudonym) Nakamoto together with the distributed digital currency Bitcoin [5]. Nakamoto proposed a novel solution to the so called double-spend problem[1] which had formed the main obstacle for the realisation of fully distributed digital currencies. Since the introduction of Bitcoin new areas of application for DLTs have been found ranging from distributed voting-systems to ledgers that keep track of the ownership of real-life assets such as diamonds and art. In this thesis the focus is on the possibilities DLTs pose on the financial structure of the Nordic capital market, with the specific goals of:

1. From a technical point of view, examine the feasibility of DLTs becoming a future integrated part of the financial infrastructure of the Nordic capital market.

2. Construct a Proof of Concept (PoC) of a DLT application for registering of securities.

There is not yet any established terminology for DLTs and sometimes the technology instead is referred to as Blockchain technologies or Distributed Consensus technologies. In this thesis the term DLTs is used as a general denomination for the collection of techniques required to keep a common distributed database with a blockchain architecture, the term blockchain is used to describe the actual structure of the database.

The rest of this thesis is structured in the following way; section 3 gives a background on DLTs as well as mentioning some of the most prominent current DLT applications. Section 4.1 to 4.7 give a technical description of DLTs and the concepts involved. Section 4.8 gives a description of a general capital market and its various actors. In section 5 the method used for obtaining the results is accounted for. Section 6.1 presents a summary of the viewpoints on DLTs of a set of actors on the Nordic capital market and section 6.2 describes the PoC application that was constructed. In section 7 conclusions are presented and in section 8 the thesis is discussed.

2.1 Limitations

For the scope of this thesis it is assumed that the data stored in a distributed ledger is meant to keep track of the ownership of assets. The means of doing this is discussed but is mostly assumed to be achieved either by storing transactions and/or storing a ledger connecting accounts and balances.

DLTs are in this thesis examined from a technical perspective. Regulatory, legal and ethical perspectives are very much relevant to examine should DLTs come to be an integrated part of the financial infrastructure but are not commented on in this thesis.

[1] In short the double-spend problem comes of the immaterial properties of a fully digitalized currency which makes it possible for users to spend the same coins twice. The problem can be solved by employing a trusted third-party such as a bank to handle the ledger, but the introduction of DLTs for the first time made it possible for a trust-less network of peers to process transactions and update a common ledger.

3 Background

The last decade's dramatic increase in e-commerce have brought on an increasing need for digital payment systems across country borders. Since the 1980s such payments have been handled by international wire transfer systems such as SWIFT[2] and various credit card companies. The introduction of internet offered new possibilities giving rise to so called Payment-as-a-Service solutions (PaaS)[3] that works as an overlay on the existing payment systems. The PaaS solutions replaces the bank at the front-end but still relies on the traditional payment and monetary systems to handle processes such as clearing and settlement. In the Nordic region inter-bank co-operations have led to increasingly efficient payment services such as the Swedish payment service Swish and its danish counter part Swipp. The introduction of distributed digital currencies, or cryptocurrencies, made it possible to also replace the back-end functionalities using DLTs. Since safe transactions are possible without the involvement of a trusted third party payments can be done bilaterally and independently of geographic location [6].

Cryptocurrencies are the original and at the time of writing the most widespread area of applications employing DLTs. While in a traditional system the payment system and the monetary system are separate, cryptocurrencies combine the aspects of a currency and a digital payment system. As opposed to a traditional currency no trusted central part is needed but instead transactions are executed over a peer-to-peer network where any willing node can validate transactions and keep track of the ledger.

To this day the most popular cryptocurrency has been Bitcoin, introduced in 2009 by Satoshi Nakamoto it was the first ever decentralized digital currency. Since the introduction of Bitcoin hundreds of cryptocurrencies based on the Bitcoin protocol have been created. A common factor for most cryptocurrencies is the use of public blockchains visible and usable by anyone, the main differing factor being the consensus process and the means to distribute new coins, all of which will be discussed in upcoming sections.

A currency is the most obvious application using DLTs, since it only involves transferring one asset type from one account to another. In the past years though more elaborate DLT applications have been created, and especially the notion of so called smart-contracts has cached the eye of many financial institutions. A smart-contract is a more complex form of transaction that allows the user to code in arbitrary constraints and triggers into a transaction. For example a blockchain storing bonds could create a smart-contract that automatically pays out periodical interest to the accounts currently owning the bond. Bitcoin partly has support for these types of transactions but only offers a simple scripting language that is not Turing-complete, i.e. there is a limitation to how complex such contracts can be. An example of a DLT-application that supports fully Turing-complete smart-contracts is the decentralized smart-contract plat-

[2]SWIFT is a standardized message service used by more than 11,000 financial institutions in more than 200 countries.

[3]Popular examples of such solutions are *Paypal* and *Mooneybookers*

form Ethereum[4], which aims to work as an underlying infrastructure for users to create their own DLT-applications on [7].

Both Ethereum and Bitcoin are using so called public ledgers where anyone can join and contribute to the network. The opposite of this, permissioned ledgers, only allow nodes that have permission to join the network. These types of ledgers are of particular interest for organisations that need to efficiently keep shared databases but for various reasons do not want to keep their ledger fully visible to the public. Many financial organisations such as banks and exchanges are currently researching these types of ledgers[5], an up- and running example is the permissioned DLT-application Linq released by Nasdaq in late 2015 that is used for private companies to keep formal records of their share holders [8].

4 Theory

4.1 High level technical description

In its essence a distributed ledger is a distributed database where the users by some consensus protocol together come to an agreement on each update. To achieve this concepts from several different areas of computer science and cryptography are utilized.

A general distributed ledger protocol dictates how a set of data structures, usually referred to as blocks, are linked together and distributed between users in a given system to form a distributed database, or blockchain, that is practically unrevisable. Each block represents the state of the database at the time the block was created, meaning that all past states of the database are preserved. Updating the data is conducted by attaching a new block to the end of the chain.

Each user holds a local copy of the blockchain, this necessitates the existence of a consensus protocol for all users to agree upon which transactions to include in the next block. The consensus protocol can be designed differently depending on the goal of the application utilizing the distributed ledger and is examined more thoroughly in section 4.7.

Each block in the blockchain has a unique identifier in the form of a hash of the block data (see section 4.2). To link the blocks together each block references its parent block using the parent blocks unique hash identifier. The links forms a chain that goes back to the first block in the blockchain, commonly referred to as the genesis block. Since the hash of the parent block is included in the block data, the hash of a block will depend on the hash of its parent block. Any change made to the data stored in a block will change the hash of that block and thus all descending blocks, meaning that in order to alter the data in a block stored within the blockchain, all descending blocks will need to be recalculated

[4]See www.ethereum.org

[5]See for example the global bank-initiative R3 (`www.r3cev.com` [accessed 2016-06-23]) and the Linux foundation open source project Hyperledger (`www.hyperledger.org` [accessed 2016-06-23])

for the blockchain to be considered valid. Depending on the procedure chosen for a user to calculate and broadcast a new block to the system this property can make it practically impossible to alter the data in the blockchain once it has enough descending blocks [9].

The data of a block can be structured in different ways depending on the purpose of the application utilizing the blockchain, but in general a list of transactions that were validated during the time-period between the creation of a block and the creation of the parent block is kept together with the hash of that transaction list. The hash of the transaction list is most commonly attained by hashing the transaction data using a merkle tree algorithm (see section 4.4).

To ensure that the owner of an account is the only one that can transfer assets from that account public-key cryptography is used. Each account is associated with a public key, to make an asset transfer a digital signature created with the corresponding private key must be included in the transaction. Thus, assuming only the owner of the account has access to the private key, the assets in the account are protected from theft. Public-key cryptography is described in more detail in section 4.3.

4.2 Cryptographic hash functions

Cryptographic hash functions are used in a wide variety of security applications such as message authentication and digital signatures. A general cryptographic hash function takes an input of variable length, often referred to as a message, and returns an output of fixed length, often referred to as a digest or hash. For a cryptographic hash function to be considered "good" it needs to satisfy the following requirements [10]:

1. Given a digest d it should be difficult to find a message m such that $d = hash(m)$.
2. Given a message m_1 it should be difficult to find a different message m_2 a such that $hash(m1) = hash(m2)$.
3. Any change in the message should result in a large change in the digest.
4. Computing the digest from the message should not be computationally heavy.

One of the most used cryptographic hash functions is the Secure Hash Algorithm (SHA), developed by National Institute of Standards and Technology (NIST) and published in 1993. Since then different versions of the SHA-algorithm have been developed, such as the SHA-256, the SHA-384 and the SHA-512, where the number indicates the size in bytes of the digest. In the Bitcoin protocol the SHA-256 algorithm is used while for example Ethereum uses the newer SHA-3 algorithm. The mathematical details of these hash algorithms will not be discussed for the scope of this thesis but for further details the reader is referred to [11].

4.3 Public-key cryptography and digital signatures

In this section the cryptographic system utilized in DLTs to provide message authentication is described. When nothing else is mentioned the section is based upon the book *Cryptography and Network Security: Principles and Practice* authored by Stalling [11].

The idea of public-key cryptography was first publicly proposed by Hellman and Diffie in 1976 [12] and later refined and implemented to use with electronic mails by Rivest, Shamir and Adleman in 1978 [13]. In a public-key cryptosystem each user is assigned two different but related keys, PR and PU, where PR is kept private and PU is made public. To encrypt a message M a user uses its private key together with a publicly known encryption algorithm, i.e. $M' = E(M, PR)$. The encrypted message M' can only be decrypted using the corresponding public key, i.e. $M = D(M', PU)$. The basic steps for sending a confidential message M between a user A and a user B using public key cryptography are then:

1. Each user A and B generates a public and a private key. The private key is kept private while the public key is kept in some public register
2. The user A uses $B's$ public key together with a publicly known encryption algorithm to encipher M. The enciphered message is then sent to the receiving user B.
3. The receiving user B decrypts the message using the same encryption algorithm but now together with $B's$ private key to retrieve the original message M. The message M is kept confidential since only $B's$ private key can decrypt the message and B is the only one with access to it.

Depending on the encryption algorithm used PR and PU can be used interchangeably for encryption and decryption. In these cases the possibility to cryptographically ensure which user has sent a message is possible, also referred to as digital signatures. The procedure for a user A to digitally sign a message M sent to a user B is much like the one for sending confidential messages with the difference that:

1. The user A uses its private key together with the encryption algorithm to encrypt M and sends the message to B
2. The user B uses $A's$ public key to decrypt the message and retrieve M, since only A has access to $A's$ private key B can be sure that it was A that sent the message.

When using digital signatures in practice it is common for storage purposes not to encrypt the entire message but rather a small bit of data that is a function of the message. In Bitcoin for example only the hash of the transaction data is encrypted to be used as the signature, not the transaction data itself. Note that in this procedure the message is not confidential since anyone having access to the encrypted message can use $A's$ public key to decrypt it.

For a public-key cryptosystem to work Hellman and Diffie put forward the following requirements on the encryption algorithm (as formulated by Stallings in [14]):

1. It is computationally easy for a party B to generate a pair (public key PU_b, private key PR_b).
2. It is computationally easy for a sender A, knowing the public key and the message to be encrypted, M, to generate the corresponding ciphertext:

$$C = E(PU_b, M)$$

3. It is computationally easy for the receiver B to decrypt the resulting ciphertext using the private key to recover the original message:

$$M = D(PR_b, C) = D[PR_b, E(PU_b, M)]$$

4. It is computationally infeasible for an adversary, knowing the public key, PU_b, to determine the private key, PR_b.
5. It is computationally infeasible for an adversary, knowing the public key, PU_b, and a ciphertext, C, to recover the original message, M.

For a cryptosystem to provide both confidentiality and digital signatures a sixth requirement is added:

6. The two keys can be applied in either order:

$$M = D[PU_b, E(PR_b, M)] = D[PR_b, E(PU_b, M)]$$

The known algorithms that have been generally accepted to fulfill these requirements are few. The previous mentioned Rivest, Shamir and Adleman was among the first to propose a viable algorithm called *RSA*, which is still today one of the most widely used public-key encryption algorithms. The algorithm used by Bitcoin (and most of the applications based on its protocol) is referred to as *elliptic curve cryptography* and was proposed independently by Koblitz and Miller in the 1980s [15][16]. For the scope of this thesis it will suffice to note that these algorithms satisfy the requirements proposed by Hellman and Daffie. For a more detailed description the reader is referred either to the cited original papers or Stalling's book on the subject [11].

4.4 Merkle trees

A merkle tree is a binary tree data structure where every non-leaf node, where a leaf node is a node without children nodes, is the hash of its children nodes. This structure results in the root node, or the top hash, being a function of all nodes in the tree. Using only the top hash the entire data structure can be efficiently verified. Figure 2 depicts a merkle tree with four leaf nodes. The leaf

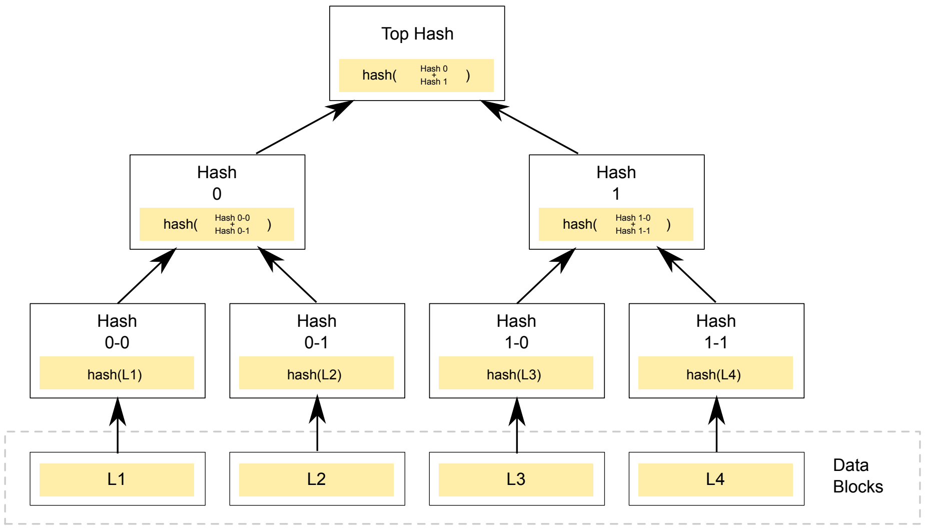

Figure 2: Four blocks of data hashed into a top hash using a merkle tree. Each non-leaf node is the hash of its two concatenated children nodes.

nodes are concatenated pair-wise and hashed using some cryptographic hash function. This is done continually until only one node is left, this is the top hash. Since the top hash is a function of all leaf nodes it can be used to quickly determine if two data sets are equal [17].

The reason for using a tree structure, and not for example just directly concatenating all the leaf nodes and hashing them, is that the tree structure provides an efficient way of determining if a specific leaf node is part of the tree. By starting at the top hash and traversing down the tree $log(n)$ number of operations are required to determine if the leaf exists in the tree, where n is the number of nodes. This can be compared to a normal hash list which requires n number of operations.

In DLTs merkle trees are used for efficient verification of blocks and transactions. In the Bitcoin protocol each block stores the top hash of all transactions stored in the block, to determine if a transaction is a part of a block the tree can be traversed as earlier described. Keeping the top hash of the transactions also allows for setting up so called light-weight nodes where a user can choose to store only meta-data of the blockchain, such as the transaction top hash, and still be able to make payments over the network.

4.5 Transaction and account based architectures

To tie the ownership of an asset to an account in a blockchain there are two popular options, *UTXO* and *account states*. UTXO is an abbreviation for *unspent transaction outputs* and is the model used by Bitcoin and many of its derivatives. In an UTXO model a transaction has an input and an output. Each input in a transaction is linked to the output of a preceding transaction. This forms a chain of transactions that must always go back to a special initial transaction created as a reward for a node that generated a new block. A transaction can have several outputs and several inputs, each linked to a specific quantity of Bitcoins. Figure 3 depicts how transactions propagates in a UTXO-based model. An initial transaction of 100 000 satoshis[6] is used as input to transaction zero. The outputs of transaction zero are used to create two new transactions, transaction one and two. These are in the same way used to create transaction three, four and five. Transaction three and six contains UTXOs that are now available to the receiving user and can be used to create new transactions.

Each unspent transaction output in the blockchain is associated with the address (public key) of the receiver of the transaction. Only the person having control of the private key linked to the address can spend the the UTXO. This means that the balance of an address is the sum of the value of the UTXO linked to the address, i.e. in a UTXO-based model no traditional accounts and balances exits, only UTXO tied to different addresses. In the UTXO-model transactions are atomic, meaning that it is not possible to only spend 40 Bitcoins of a transaction output worth 100 Bitcoins. In this case two transaction would need to be created, one spending the 40 Bitcoins and another sending the change of 60 Bitcoins back to oneself [18].

In an account based model the state of the blockchain is made up of account-addresses tied to balances, i.e. a ledger. This model is used by, for example, Ethereum. In the current version of Ethereum both an account-state and transactions are stored in the blocks, with the difference that the transactions does not contain inputs and outputs.

Whether an UTXO or account based model should be used depends on the aim of the application. In [19] Ethereum's co-founder Vitalik Buterin puts forward the arguments for an UTXO-based model as having benefits both with respect to scalability and privacy, especially if a new key-pair is generated for each transaction. An account based model on the other hand Buterin argues saves disk-space since each transaction only needs to have one signature and one receiver (output). An account based model also offers a more efficient way to determine the the balance of an account as well as being overall simpler and easier to understand.

A problem with account-based models are that they do not offer a built-in way to counter replay attacks, i.e. attacks where an already processed transaction is re-sent over the network in an attempt to steal funds. In an UTXO based model this is not possible since the re-sent transaction would now point towards a spent transaction output and not the required UTXO. To counter this for example Ethereum has chosen to associate what they refer to as a "nonce" with each transaction and account. The nonce is simply an integer that works as a counter. When an account sends a transaction the accounts current value of the nonce is put in the transaction data. Each time a transaction is sent from the account the nonce in the account is incremented, meaning that if someone

[6]One Satoshi is the smallest nominee of value in Bitcoin.

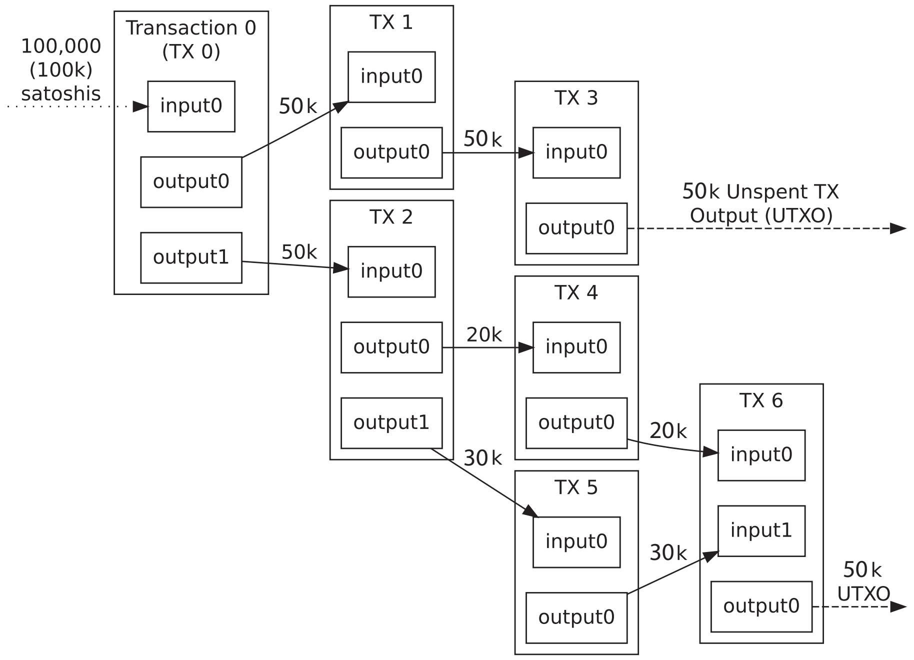

Figure 3: A schematic view of how transactions are linked to each other in the UTXO model as used by Bitcoin. *source: bitcoin.org/en/developer-guide*

were to try to resend the transaction now the nonce in the transaction and the account would not agree, rendering the transaction invalid.

4.6 Permissioned and public ledgers

A public ledger puts no constraint on who is allowed to read and who is allowed to write to the ledger. Anyone is allowed to participate in the consensus process where it is agreed upon how to define the next block that is to be attached to the end of the blockchain. Such a blockchain is considered to be fully decentralized and examples of such applications are Bitcoin and Ethereum. The consensus protocol in an public ledger cannot assume any trust among nodes and thus need to be based on a crypto-economic process such as proof-of-work or proof-of-stake since no trust can be assumed among the users.

Buterin writes in [20] that permissioned ledgers can be distinguished into two categories, fully permissioned ledgers and consortium ledgers. In both cases the user base is permissioned to a predetermined set of users where, to varying degrees, some trust is assumed among the users. In a consortium ledger the system can be said to be partially centralized meaning that there is not a single node that on its own controls the consensus-process, but rather a set of permitted nodes that together validates the next block to be attached to the blockchain. The read- and write-privileges can be set either to be fully public or to some combination where for example anyone can read from the blockchain but only a certain set of nodes can write to it. In a fully private ledger write-permissions and block-creation privileges are restricted to one single node, where as read-permissions can be set to be either fully public or permissioned.

Depending on the circumstances public and permissioned ledgers have their advantages and disadvantages. A public ledger is not controlled by any single entity and requires no trust among the users, thus reducing the risk for miss-use by the controlling part. The users are also in some way protected from the developers since it is up to the system as a whole to accept and start to use new software. The openness of the public blockchain also makes way for a system of many independent users, which in itself increases the safety of the system since the risk of both data loss and hostile takeovers decreases as the number of users increase. Since no one single-handedly controls a public ledger there is no efficient way to revert transactions, alter balances or quickly change the rules of the system, this aspect can be seen as an advantage in for example the use case of cryptocurrencies, but for example for a consortium of banks who want to share ledgers among each other, giving up this control is usually not acceptable. In these cases a permissioned ledger solution is arguably more viable.

4.7 Consensus

A consensus protocol is the procedure used to ensure that all nodes in a network apply the same set of transactions to the ledger at each update. It is a fundamental mechanism that can be designed in different ways depending on the desired properties of the application. What type of consensus protocol is used highly affects the scalability and efficiency of the application. A main discriminating factor among different protocols is that of a protocol being *probabilistic* or *deterministic*. In a probabilistic protocol all nodes in the network are trying to generate the next block (commonly referred to as mining).[7] What determines each node's individual probability to succeed with generating the next block, and thus its influence over the blockchain, is usually the differing factor among different probabilistic protocols. In a non-probabilistic consensus protocol instead a deterministic process is used to determine what transactions to include in the next block, meaning that it is possible to predict which node that will generate the next block.

When using a probabilistic protocol two nodes could potentially generate a block at roughly the same time. This leads to a so called fork in the blockchain where different parts of the network accept different blocks. In probabilistic consensus protocols the forking problem is usually solved by having a rule that says that a node should always and only work on the fork that has the greatest work in it. This means that as soon as one fork generates a new block all

[7]This is not entirely true. In many distributed ledger networks there exists so called light weight nodes that do not participate in the mining process but still are able to create and receive transactions.

nodes in the network will switch to that fork and the blockchain will be joined again. The blocks in the abandoned fork will be discarded, meaning that the transactions that only existed in the discarded blocks will be reversed.

In a probabilistic protocol a user cannot be entirely certain that a transaction conducted is not going to be discarded due to forking. The risk of this happening reduces as more blocks are created and the transaction is deeper into the blockchain, and it is therefore in for example the Bitcoin network recommended to wait at least six blocks to be sure that a transaction is final. This property is usually referred to as *block finalization.* Deterministic protocols normally have instant block finalization while probabilistic protocols can only talk about block finalization in terms of probabilities.

4.7.1 Proof of Work

Perhaps the most intuitive consensus-process is for each user to be delegated one vote and let all users vote on what transactions to be included in the next block. The set of transactions with the most number of votes are then included. The problem with this type of system is that it is vulnerable to a so called Sybil-attack, where a user can create many accounts and thereby gain a disproportionate amount of influence over the ledger. The Bitcoin creator Nakamoto solved this problem by instead basing a user's amount of influence on a physical quantity, namely the amount of energy in the form of CPU-power a user has access to. This type of consensus-process is generally referred to as Proof-of-Work (PoW).

The general idea of PoW was not introduced with Bitcoin but was first proposed by Dwork and Naor in [21] as a means to combat junk-mail by attaching a cost in the form of CPU-power to the action of sending messages over a network. This way sending a single message would not affect a normal user while sending large quantities of messages would be costly. The notion 'Proof-of-Work' was later formalized by Jakobsson in [22].

PoW as a consensus-process is to a large extent based on Back's idea of hashcash [23]. In a peer-to-peer network making use of a blockchain to share data the work is attached to the action of broadcasting a new block to the network. The work is done by computing what is usually referred to as a token, or nonce, by the use of a *cost-function.* Hashcash introduced a PoW-system using a cost function that is:

- *Non-interactive*, meaning that each node can start to compute the nonce without additional information from peers.
- *Trap-door free*, meaning that no node has any advantage over another in terms of computing the nonce.
- *Publicly auditable*, meaning that without the use of any trapdoor or secret information it can be efficiently verified by any third party that the work has been performed.

- *Unbounded* with respect to the *probabilistic cost* of computing the nonce, i.e. in theory there is no upper bound to how long time it can take to compute the nonce, even though the probability of not finding the correct nonce rapidly goes towards zero as the average time is passed.

The Bitcoin-network (and many other) makes use of the hashcash protocol by using SHA-256 as the cost function. To control the difficulty of the cost function it is required that the digest is below a certain threshold. This makes it possible to control with which average frequency a new block is created by adjusting the difficulty according to the total computing power of the network. In practice each node tries to find a hash that is below the threshold by altering data from the latest block and using it as the digest to the cost-function. The most efficient known way to compute this is by brute force, which makes it possible for any node to instantly verify that a claimed work has been performed. As mentioned earlier this practice is commonly referred to as mining since computing a valid digest comes with a cash reward.

In a public network where no trust and no synchrony can be assumed the PoW consensus process has been empirically proven to be both safe and robust during the years it has been employed in the Bitcoin-network. As the total computing power of the network increases the safety of the blockchain increases since it becomes increasingly harder for a single entity to gain control over the network.[8] A common critique of PoW is the significant cost-overhead brought on by the large amounts of electricity consumed by nodes competing with each other over mining rewards. It has for example been estimated that the Bitcoin-network consumes around 215 MW of energy, resulting in an energy consumption per transaction equal to that of 1.57 American households daily consumption [24].

4.7.2 Proof of Stake

In an attempt to further increase the protection against attackers and to reduce the large amount of power consumption required by a PoW-based consensus processes, the idea of Proof of Stake (PoS) evolved in the Bitcoin developer community. Instead of, like in a PoW-system, having a node's computational power govern its influence on the ledger, a nodes total holdings recorded in the ledger decides its proportion of influence. The idea builds on the assumption that a node holding a large stake in the blockchain is more willing to maintain the trust in the blockchain and thus more likely to operate as a friendly node. Bitcoin also shows that a PoS-system would reduce the risk of an entity gaining monopoly over the blockchain, since acquiring a majority of Bitcoins would be far more expensive than to invest in enough hardware to gain more than 50 percent of the total computational power of the network. [25]

To prevent users of transferring holdings between nodes with the purpose of manipulating the consensus process the concept of *coin age* is used in many

[8]This notion has been somewhat disturbed by the introduction of mining-pools currently controlling large portions of the total computing power in the Bitcoin-network. For more information see `https://bitcoinchain.com/pools` [accessed 2016-06-23]

PoS-systems. In a system employing the coin age concept, a coins influence-contribution is dependent on the time it has been consequently held by a node.[9]

Another issue necessary to address in PoS-systems is the low cost of forking the blockchain. Since almost no computational work is attached to the process of creating a block the incentive for villainous nodes to fork the blockchain in for example double-spend attacks, or simply for greedy nodes to work on all forks to not miss out on block rewards, increases. To combat this many PoS-applications have introduced the concept of *check-point blocks*. Blocks created before a check-point block can no longer be revised and therefore all transactions in those blocks are safe from double-spend attacks [26][27].

4.7.3 Practical Byzantine Fault Tolerance

The notion of Byzantine failure was first introduced by Lamport, Shostak and Pease to describe the problem of arbitrary behaviour of nodes in networks [28]. The problem is illustrated by a group of Byzantine generals that need to reach a majority agreement if and when to attack a surrounded enemy city. The generals can only communicate with the use of messengers and an unknown set of generals are potentially traitors that will try to spread disinformation. Lamport et al. proved that the honest generals can only succeed if at least $3f + 1$ number of generals are honest, where f is the number of traitorous generals. The protocol presented for achieving this was not practically feasible though since it assumed a synchronous network.

In [29] Castro and Liskov introduced the Practical Byzantine Fault Tolerance (PBFT) protocol. With the use of effective message authentication codes it was shown that the PBFT protocol realises the $3f+1$ lower bound in asynchronous networks. Since then improvements have been made and many variants of PBFT protocols have been developed, although the general principle is the same. For the sake of this thesis only a simplified overview of a general PBFT protocol will be made, for a more extensive description the reader is referred to [29].

In a PBFT-protocol one node is automatically assigned a special status known as the primary node. For all requests, for example a transaction, to be processed in the same order the primary node assigns each request a number indicating its turn of execution. The primary node then broadcasts the request to all other nodes which in their turn re-broadcast the request to all other nodes. Only if a node has received a sufficient number of identical requests from the rest of the network it will broadcast a so called commit-message, indicating that it accepts the request. This way the network makes sure that the primary node has not broadcast two conflicting requests. If not enough identical requests can be collected it means that the primary node in some way has stopped functioning correctly. In this case a new primary node is selected according to a set of pre-determined rules.

[9]See for example *Peercoin* (`https://www.peercoin.net/` [accessed 2016-06-23]) that bases an account's likelihood to create a new block on the product of the number of coins held and the number of days each coins has been held, and *Nxt* (`https://nxt.org/` [accessed 2016-06-23]) that only allows coins held longer than 1440 blocks to contribute to a node's influence.

As opposed to the PoW- and PoS-protocol a PBFT-protocol requires each node to have knowledge about all nodes currently participating in the network. It is therefore arguably not suitable for a public ledger application where it cannot be assumed that all nodes know of all nodes. In a permissioned ledger application though a PBFT-protocol could offer several advantages over both PoW and PoS. A PBFT-protocol does not need to handle the issue of forking since block-finalization is instant. This makes it possible to greatly increase the possible transaction-volume compared to that of the probability-based protocols. The major concern regarding PBFT-protocols is the scalability with respect to the number of nodes. Applications currently utilizing PBFT-protocols are file-sharing systems usually consisting of no more than 10-20 nodes. To be used in networks consisting of several thousands of nodes such as the Bitcoin network more research needs to be done.

4.7.4 Other protocols

Apart from the previously mentioned protocols there are currently a large variety of other consensus protocols being experimented with. Some applications use combinations of PoW and PoS and others, such as the transaction network Ripple, uses a kind of deterministic voting system for the network to agree on what transactions to include in the next block [30].

4.8 The securities marketplace

This sections defines actors and processes on the general capital market and is to a large extend based on [31].

A capital market is a financial market where equity and debt are traded in order to transfer wealth from savers to people or companies that can put it into productive use. Equity represents ownership in a company and is also often referred to as stocks or shares, while debt reflects the borrowing of investors to the debt-issuer often conceived by the issuing of bonds.

4.8.1 Participators

The participators of a general capital market are numerous and can be categorized in several ways but for the purpose of this thesis we will define them as:

Investors - individuals or institutions that buy equity or bonds, where an individual investor is a private person and institutional investors are for example pension funds, insurance companies or mutual funds.

Security trading organisations (STOs) - STOs are organisations that in some way trade with securities. The trade can be conducted both for their own sake in an attempt to buy low and sell high, but can also be done at the assignment of an investor, in this case the investor is often of the institutional kind. An STO can own securities on their own and either make direct trades with agents, investors, other STOs or trade over an exchange. Usually the STO is a sub-organisation of a larger financial company such as a bank.

Agents - the usual means for an individual investor to act on the capital market is to go through an agent which will, in exchange for a fee, act as an intermediary between the investor and an STO or exchange. Examples of agents can for example be stock brokers, Internet brokers and financial advisers.

Regulators - in order to make sure that business is conducted according to laws and regulations regulators monitor the markets. Usually all transactions undertaken by an STO need to be reported to a regulator.

Custodians - Custodians keeps “custody” of securities, i.e. they are the organisations that keeps register of ownership of securities.

Figure 4 demonstrates the role of the different actors as a transaction is executed between two individual investors. The figure depicts an investor A and an investor B who want to sell and buy a particular stock respectively. In this case the two investors are clients to two different agents, for example two different Internet brokers, which they contact to put in the sell and buy orders. As the agents receive the orders they forward it to the exchange where, assuming the buy and sell price match, the buyer and seller will be paired and the agents will be informed that a match has been made together with the relevant information about the respective counterpart. In order to actually execute the exchange of the stocks and the money each agent needs to instruct one or several custodians to settle the trade. This process of settlement will be discussed in more detail in the upcoming chapters since it is likely to be highly affected in a blockchain based model. As the trade is settled information about the transaction needs to be forwarded to regulators and a receipt is sent to both investors. At this point the trade is considered settled.

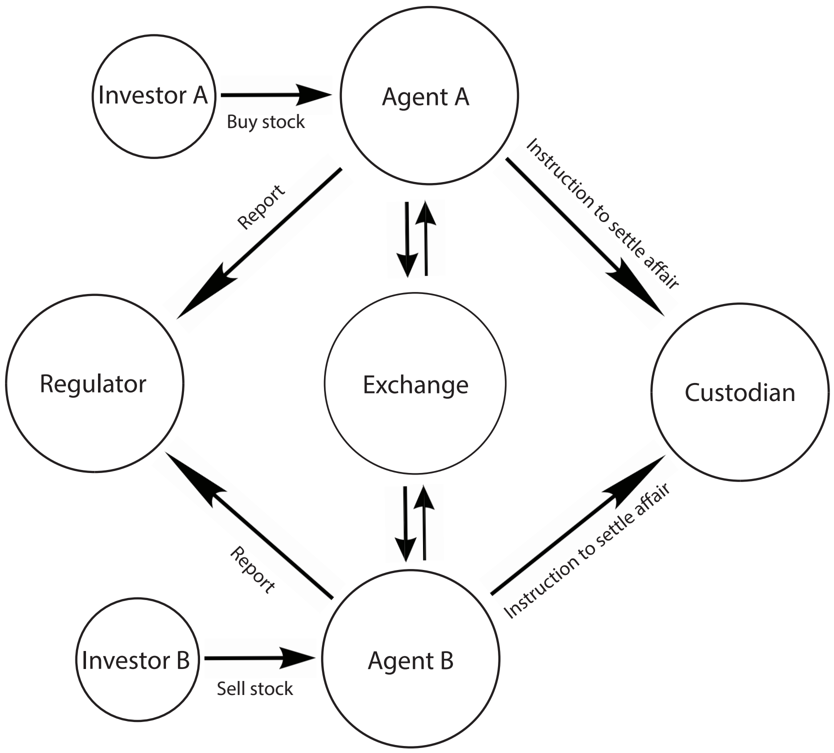

Figure 4: A scheme of two individual investors making an trade through two different agents.

4.8.2 Internal structure of an STO

The most common way to structure an STO is to structure the different working areas into the front-, middle- and back-office. The front-office can be seen as the window against the market, here trades are conducted and contact is held with clients and costumers. The middle-office supports the front-office in a logistical manner, performing tasks such as data input into trade-systems, agreeing upon trade details with counter parties as well as conducting risk analysis and trade surveillance. The back-office settles trades and maintains the STO's formal books and registers.

4.8.3 The trade life cycle

The trade life cycle refers to the entire process from where an order is created to where the trade is settled. The trade life cycle involves a number of steps taken in order to minimize errors and financial risk. Historically most of the steps in the cycle have been carried out manually, leading to the re-entering of the trade-data into several different registers thus increasing the risk of manual errors. Today the securities industry largely aims to achieve so called straight

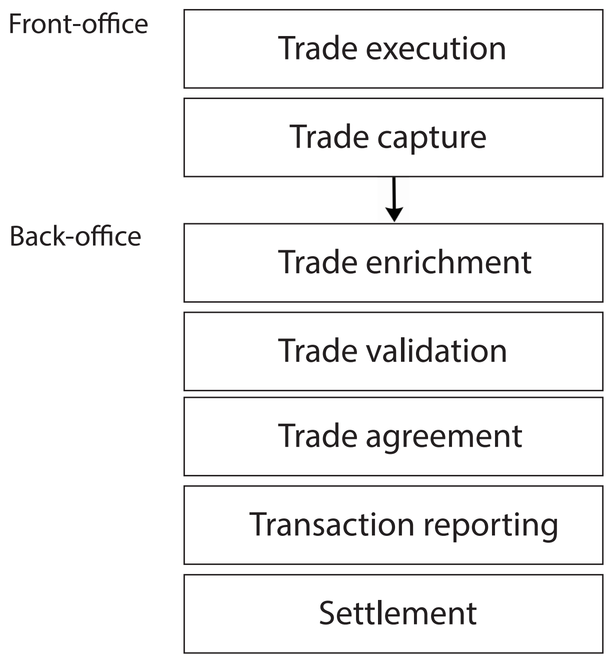

Figure 5: The life cycle of a trade as seen from a general STOs point of view.

through processing (STP), which basically means that all the steps in the cycle should be handled in an automated fashion.

Figure 5 depicts the main steps in a trade life cycle from a general STOs perspective. The trade life cycle begins as a trade is executed in the front office, this means that a seller and a buyer have been matched and agreed to trade. As the match and agreement have been made details regarding the trade have to be formally recorded, or captured, within the organisation and conveyed to the back-office where first a so called trade enrichment takes place, attaching further data such as custodian details and method of transmission, to the trade. After this a validation usually is conducted where the trade data is checked against certain constraints in order to minimize errors before the trade is displayed to the outside world. Having successfully validated the trade a check is made with the counterpart ensuring that both parties agree on the trade data.

Having attached and validated all the relevant data to the trade, it is usually reported to regulators and the settlement process begins. As earlier mentioned the process of settling a trade is defined as the actual exchange of cash and securities, often done at the custodian. The date for settlement is agreed upon by the trading parties and is usually referred to as the value-date. At this date both parties needs to deliver the agreed upon cash and securities in order for the trade to settle. The time between the execution of a trade and the value-date varies between countries and product groups, in western Europe the standard time for stocks is decided upon to be two days. Part of the reason for having this time-gap between trade-execution and settlement is that a considerable part of the trades conducted today are between parties that have their location in widely different parts of the world with the consequence of the involvement of a large number of intermediaries and several custodians in order for the trade to take place.

In order for the trade to settle the custodian will need confirmation from both parties of the trade and to minimize financial risk settlements are usually aimed to be done on a so called delivery vs. payment basis (DvP), where the change of cash and securities are made simultaneously. The opposite of DvP is so called free of payment (FoP) where one of the parties deliver before the other. Which payment method to use is agreed upon by the trading parties and forwarded to the custodian. Settling trades in today's global market is an intricate business and further treatment of the process, such as settlement failure and the inner workings of custodians, won't be treated for the sake of this thesis. The important thing to keep in mind is that processing settlements in the high volumes of today's markets require updates and syncing of many registers and usually takes several work-days to complete.

4.8.4 Central counterparties

A CCP is an organisation that acts as a an intermediary between the buyer and the seller of a trade. By taking the part of both the buyer and the seller the CCP guarantees to deliver the agreed upon cash and securities at value-date, thus decreasing the so called counter-party risk where both parties risk the default of the other. Another reason for using a CCP is the reduction of the number of settlements needed. In a process referred to as clearing the CCP can net the trades executed during some period of time and only settle the netted amount for each member. Figure 6 depicts an example where several trades in the same security is done with the help of a CCP, instead of settling each trade individually, the CCP can clear the trades and only settle the netted amount at the custodian.

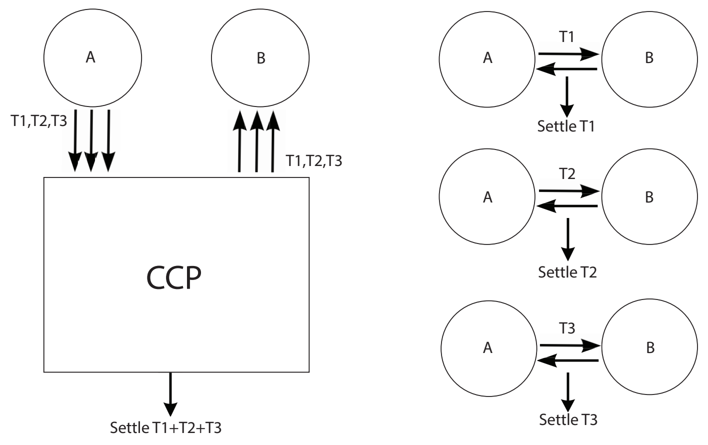

Figure 6: To the left: Three transactions made using a CCP. All transactions are netted against each other and settled only once. To the right: The same transactions done without the use of a CCP, all transactions are settled individually.

5 Method

The methods for reaching goal one of the thesis consist of literature studies and the interviewing of various actors on the Nordic capital market. The interviews conducted were so called *qualitative semi structured interviews*. This means that, as opposed to a structured interview, there were no hard set of questions asked to all interviewees in a set order [32]. Ahead of each interview a set of questions were prepared and general information about the institution that were to be interviewed was researched. The prepared questions were used as guidelines for the interview, as the interviews progressed some were removed and some were added. All interviews took place in person at the respective interviewee.

The reasons for choosing a qualitative approach rather than a structured quantitative approach for the interviews are:

1. All interviewees are coming from different institutions and thereby have varying knowledge and perspective on the technology. Flexibility with respect to the questions being asked is therefore desirable.

2. The number of interviewed persons are few, but their knowledge and insight of their respective area is great. Therefore it is more reasonable to base conclusions on their individual opinions rather than the patterns of a small data-set.

The DLT application for registering and transferring ownership of securities was coded in Java. The application was built from the ground up with the exception of the cryptography, which is from the Java implementation of the open-source project Ethereum.

6 Result

6.1 Actors on the Nordic capital markets view on DLTs

In this section the view on DLTs of various actors on the Nordic capital market are presented. The views have mainly been collected by the means of interviewing relevant people employed by the respective companies. The interviews were conducted in person and recorded. The views are mostly presented by paraphrasing the interviewees, when exact quotes are made these are in quotation marks. Note that most of the interviewed actors have not currently expressed any official views on DLTs, the opinions expressed represent those of the interviewee and not their employer. Some actors that were interviewed (The Swedish Securities Dealers Association and the Financial Inspection) had a regulatory perspective but did not express any particular opinions about the technology from a technical perspective, and are thus not included in this report[10]. In some cases actors have released official reports presenting their view, when views from a report are presented the report is clearly cited.

6.1.1 Avanza

Avanza Bank is a Swedish Internet broker and digital savings platform that provide financial services to both private persons and companies. They have an outspoken focus in providing their costumers the best user-experience on the market and have the last decade come to be the leading Swedish bank in terms of securities trading for private persons. Peter Strömberg works as Chief Information Officer (CIO) at Avanza Bank and has an extensive background in finance as well as in IT.

When asked about his general view on DLTs Strömberg says that the technology could likely be applied in any area where ownership needs to be registered in a safe and efficient way. In the financial industry the most obvious application area is that of streamlining the capital market post-trade. The initiative to examine how these areas of the market can be made more efficient is a welcomed one from Strömberg's perspective. The last decades trade-execution and trade-matching have been made dramatically more efficient and costs have been decimated while the post-trade area has been largely at a stand-still.

Strömberg says that the post-trade system needs to be efficient in terms of of trust, speed and cost and that DLTs are likely to have many advantages in these areas compared to the current system. The possibility of atomic transactions[11] where no CCP is needed could dramatically decrease the time and cost

[10]For more details on DLTs in the Nordic capital market from a regulatory perspective see [33].

[11]i.e., transactions where money and securities are swapped in one single step.

of transactions. For Avanza Bank this could in its turn cut costs and allow them to offer cheaper securities trading to their costumers.

Strömberg believes that one of the greatest obstacles for DLTs to become a reality is for banks and other actors on the financial market to find a common standard. Strömberg says that there is good ground to believe that this will succeed though. A good example is the common payment service Swish and bankID that many of the largest Swedish banks successfully developed together.

Strömberg sees that there is a high probability of many DLT-applications emerging in the coming years. As for now Avanza Bank are not looking into starting any major DLT-projects on their own. They are though following the development of the technology closely and are open for co-operations if the opportunity should present itself.

6.1.2 Nasdaq OMX

Nasdaq owns and operates stock exchanges in the United States and in eight European countries including Denmark, Sweden and Finland where also post trade operations are performed. They are also a provider of financial technology and services to other market operators around the globe, altogether serving more than 100 marketplaces with their in-house developed technology. Johan Toll works as product manager and business developer at Nasdaq focusing on new technologies such as blockchain and machine learning, based in Stockholm. When asked about his general opinion on DLTs potential in the financial industry he says that DLTs could have a large impact on the industry the coming years. He sees obvious applications for DLTs in primarily the post-trade area where custody, clearing and settlement could be made more effective with the use of shared ledger technologies. A scenario like this would potentially allow exchanges to directly handle tasks in the post-trade value chain that is now handled by actors such as the CSD. Smart-contracts is another promising area heavily linked to the DLT technology. Creating automatically executed contracts could automate many processes that are today handled manually and has potential to cut costs in many areas.

Toll also sees potential in more independent applications. For example Nasdaq has already released a DLT-based application called Linq which is used to register ownership of shares in private companies. Linq currently uses a permissioned ledger deployment where Nasdaq controls which users are allowed to connect to the network. When asked what advantages Linq has over for example a central database Toll mentions the immutability of a distributed ledger, built-in settlement capabilities as well as the shared accessibility of the ledger and it's transaction history. It is also a cost-efficient way to keep the database synced and updated.

In order to replace the current post-trade systems with DLTs Toll says that the focus has shifted from a technological focus into looking how the full ecosystem with market participants, existing processes and regulatory framework will be affected by this new approach. Through the move into permissioned ledgers, performance, data entitlements and control of user access can be much easier adapted for the financial market needs. Introducing permissioned ledgers opens up for more efficient consensus-processes as well as allowing the users to retain control over the transparency of the ledger. This is therefore a likely way for the industry to go in the coming years.

Toll is generally optimistic of DLT's future use in the financial industry. He believe that they have the potential to affect the industry in many areas as well as acting as a catalyst for general research intended to streamline the post-trade area. Nasdaq have already released DLT-based applications and are currently actively researching and developing the technology and its business implications further. According to Toll it is likely that permissioned ledgers will play a major role since they increase the speed of settlement while also reducing the capital risk, combined with the ability to scale the technology in terms of transaction volume as well as allowing the users to control the transparency of the ledger.

6.1.3 Cryex

Cryex is a centrally cleared currency exchange resided in Stockholm. In the near future they are planning to provide an FX trading-platform with the possibility to trade in both fiat- and cryptocurrencies. Anders Stenkrona is the head of board of Cryex and has an extensive background in risk calculation.

When asked what potential DLTs have to create a more efficient capital market Stenkrona says that there is indeed a potential for DLTs to do this, especially in the post-trade area. Still there is some doubt that banks will currently commit to these investments. Recent regulations have already forced banks to make large investments in their IT-systems and at the moment their focus is likely to lie on increasing their revenue rather than decreasing their costs. Stenkrona believes that these types of investments are probable to happen in the future though.

According to Stenkrona it is necessary for the industry to find a common standard if the technology is to reach its full potential. One of the main technical difficulties for DLTs to become an integrated part of the financial infrastructure is likely to lie in the standardization process. A way to efficiently handle large transaction volumes will need to be developed as well. Today's cryptocurrency-networks have proven to be robust but to work as an integrated part of the financial infrastructure much larger transactions volumes will need to be handled than is currently possible in these networks. A permissioned ledger opens up for more efficient ways of handling consensus and is therefore a possible solution according to Stenkrona. A permissioned ledger allows for better control of who is allowed to access the ledger as well. This makes it more likely to be accepted by regulators than if a public ledger were to be used. The R3-project where banks are examining how to share data using a permissioned ledger together with an efficient consensus process is a good example of a plausible future solution.

Stenkrona thinks that there will be a demand for fully public ledgers in the future as well. As we are moving towards a cash-less society open cryptocurrencies could offer functionalities such as anonymous payments. The need might be reduced though as institutions adopt DLTs and can offer the same seamless transactions as the Bitcoin-network can today.

Stenkrona says that Cryex is currently applying to get permission from the Swedish FSA to conduct its business. As for today the interest in cryptocurrencies from institutional investors is limited. Reasons for this is for example that the market cap of Bitcoins and other cryptocurrencies is to low for global institutional investors to be able to make investments of meaningful size. Some institutions are also sceptical about the safety and legality of cryptocurrencies. This is nothing new when introducing new technologies though and Stenkrona believes that the scepticism will fade with time as the technology is adopted and better understood by the institutions.

6.1.4 SEB

Skandinaviska Enskilda Banken (SEB) is a Swedish bank that operates globally, with emphasis in the Nordic region. As one of the largest banks in Sweden they have an intrinsic need for robust and efficient computer systems. Anders Nyqvist works as chief strategiest at SEBs CIO office with responsibilities such as scouting current trends in financial technology and evolving strategies for SEBs future technology use.

SEB are currently in an experimental phase regarding DLTs. They are involved in the international DLT-cooperation DLG as well as developing their own Proof of Concepts. When Nyqvist is asked about his general view of DLTs he says that there are many different areas of applications in the financial industry. An example is improved data-sharing between banks, where for example Know your Costumer-processes could be made a lot more efficient if banks shared more registers. There is also a potential to improve most steps in the capital market value-chain. If ownership of securities were registered on a blockchain, sellers and buyers could easily find each other and the process of clearing and settling could be made in on single step without the involvement of any third part. A DLT-based system could also make it easier to report to regulators by just giving them access to the blockchain where transactions etc. are stored.

Nyqvist says that Smart-contracts is another area where DLTs could potentially increase efficiency and straight through processing in the banks internal systems. An example is the case where large companies sets up an account at SEB. These accounts usually has the need to handle complex payments and several types of currencies. In a DLT-based system much of this work could be handled automatically with the use of code stored and executed in the blockchain, i.e. a smart-contract.

For DLTs to be functional with a bank the size of SEB they will need to be improved to handle larger transactions volumes than is currently possible. For this to happen Nyqvist sees a need for a more efficient consensus-process than the PoW-scheme currently employed by many of the largest DLT-applications. The PoW-scheme as it is used today is not sustainable in the long term due to its vast energy-consumption and bad scaling. Nyqvist says that one way of creating more efficient consensus-processes could be to use permissioned ledgers where the value of the currency is coming from the guarantee of the participating banks rather than the work put in by the nodes, as in a PoW-schedule.

From a bank such as SEBs perspective Nyqvist sees a potential for the technology to improve the process with which banks share information with each other. Smart-contracts are highly interesting since they have the potential to automate many processes in the back-office that are today conducted manually. If an implementation of DLTs is to happen in the banks internal systems Nyqvist thinks that it is not likely to happen in the next few years though. The systems used by large banks today are integrated and dependent on each other and can sometimes be several decades old. Making extensive changes to these are therefore not done easily.

6.1.5 Euroclear

Euroclear operates CSDs in several European countries there among Sweden and Finland. They provide financial services such as transaction settlement and record-keeping of different asset-types such as bonds, equities and funds. In spring 2016 Euroclear released [3] where their view on DLTs potential use in capital markets are presented. The report states that DLTs could provide “major operational benefits for users”. Data could be transferred and updated among the market-participants in near real-time, removing the need for data enrichment[12]. DLTs could also increase trust among participants since it would be trivial to prove your ownership of an asset by just having your counterpart control its version of the ledger. Increased trust would reduce the credit-risk among participants and allow capital to be utilized in a more effective way. Euroclear also sees a potential to streamline the settlement-process by an instant exchange of cash and securities, thereby removing many steps such as central clearing and post-trade affirmation.

The report mentions that there is indeed ways of reaching these benefits with already existing technology. Instead of each participant having their own local version of the ledger, a central database could be provided by an extended version of a CSD. The participants could then query the database to retrieve and update data. With DLTs though there would be no single point of failure and the participants could bilaterally choose to reveal information to each other without queering a central authority. DLTs also opens up for complex forms of smart-contracts that is not possible with a traditional central database. Having the database distributed and cryptographically secured also decreases the risk of malicious data manipulation.

Euroclear sees that a scenario where DLTs are fully integrated in the capital market infrastructure, i.e. all assets as well as cash are kept on blockchains, would affect many of the actors on the capital market. In cash-asset transactions the need for CCPs would diminish since settlement would occur almost instantly and the trading parties would be able to assert that the opposing party currently owns the cash or asset. Private traders would likely be affected as well since

[12]A transaction often affects several databases, aligning these is referred to as data enrichment.

the frequency with which trading could be conducted would be limited.Traders would have to wait until a new block has been created and their ownership of the asset is registered in the blockchain before they can sell the asset again. This could potentially limit the possibilities to conduct the algorithmic high frequency trading that has been increasingly more popular in the capital markets the last few years. The role of the CSD would likely shift to be more service oriented rather than that of a custodian. According to Euroclear there would still be a need for a coordinated oversight of asset issuance and the development and drift of the blockchain.

For DLTs to be able to replace core-parts of the financial infrastructure Euroclear sees that the questions over the scalability of the technology need to be answered. Even though major improvements have been made since the original Bitcoin-protocol it is still unclear whether DLTs could support the large data volumes seen in the capital markets. The industry will also have to agree on how to design the protocols, such as if a permissioned or public ledger should be used, how to make the ledgers interoperable with other distributed ledgers as well as older systems and how the consensus process should be designed. They also see a risk in the managing of anonymity, the security of public-key cryptography is dependent on keeping the private key safe. If a private key were to be compromised sensitive information could end up in the wrong hands.

6.1.6 Summary

Table 1 displays a summary of the views of the interviewed persons.

	Avanza	Nasdaq OMX	SEB	Cryex
General attitude towards DLTs	Positive	Positive	Positive	Positive
Own development	No	Yes (released products)	Yes (PoC's, co-operations)	No
Most interesting application area	Post trade	Post trade, smart contracts, independent applications	Inter bank data-sharing, smart contracts, post trade	cryptocurrencies post trade
Biggest obstacle	Standardization	Standardization, performance	Performance, integration	Standardization, performance

Table 1: A summary of the views on DLTs of the interviewed persons.

6.2 Proof of Concept

A permissioned distributed ledger PoC was built to examine the technical difficulty of developing DLT-applications as well as examining how registration of securities could be handled in a private distributed ledger. The PoC was coded in Java using the Eclipse IDE.

6.2.1 Requirements

The PoC need to fulfill the following requirements:

R1. A node shall be able to store accounts.

R2. An account shall be able to have a balance of assets.

R3. Only permitted nodes should be able to connect to the network.

R4. Nodes shall be able to transfer assets to other nodes.

R5. Public-key cryptography shall be used for user- and message-authentication.

R6. Nodes shall be able to create assets.

R7. Altering an asset shall be limited to the creating node.

Where a node is an instance running the PoC.

6.2.2 Specifications

To fulfill the respective requirement the following list of specifications needs to be implemented (S1.1 and S1.2 belongs to R1 etc.):

S1.1. A node should store account files.

S1.2. Account files should contain an accounts private and public key.

S2.1 A node should store block files.

S2.2 A block file should contain a map linking an accounts public key to that accounts current holdings.

S3.1. A block file should contain the IP-addresses of all permitted nodes.

S3.2. To accept an incoming connection each node should first confirm that the senders IP-address exists in the latest block.

S4. There should exist a transaction data structure for transacting assets between accounts.

S5.1. A transaction should contain the digital signature of the sender.

S5.2. Each node receiving a transaction should verify that the transactions digital signature is valid.

S6.1. There should exist an asset-type data structure.

S6.2. There should exist a transaction type that creates new assets types.

S7.1. Each asset type should store the public key of the account that created the asset.

S7.2. To alter an asset type a digital signature matching the public key stored in the asset type must be provided.

6.2.3 Design

This section describes the design of the PoC. The section is meant to give the reader a general overview of the design hence not all details are included in the description. For a fully detailed description the reader is referred to the source-code in appendix REF. The section starts with describing the data structures of the PoC and the goes on to give a description of the most important methods and processes. All methods and classes are written in `typewriter-style` to facilitate easier reading.

Structure and classes

The PoC is structured into five packages:

- **Core** - Contains the main data structures such as `Block` and `Transaction`.
- **Crypto** - Contains cryptographic classes needed for hashing and public-key cryptography.
- **Network** - Contains the client- and server-side code as well as code for blockchain synchronization between peers.
- **Util** - Contains various utility classes used by the other packages.
- **Start** - Contains the code used to set up users and start the program.

Core Figure 7 depicts a UML class-diagram of the core package. Following is a description of the core classes:[13]

`Blockchain` The interface towards the blocks that are stored on the local machine. The Blockchain class does not itself have any members of the type `Block`. The diagram shows three fields in `Blockchain`: height, directory and synchronized. The directory field is the relative path to the folder where the blockchain is stored, the height field is the current height of the blockchain and the synchronized field is a flag indicating which peers the blockchain is synchronized against.

`Block` The basic data structure making up the blockchain. Each block has a list of transactions, a map of public keys to account states, a list of valid IP-addresses, a time-stamp and a hash of the data of the preceding block.

`AssetType` The data structure representing the different types of assets that can be owned by a user. Each asset type has a name, a unique identifier and a classification (such as stock, bond etc.). The account that creates an asset type owns that asset type. This is maintained by storing the public key of the creating account in each asset type.

[13]Note that the diagram is meant to give the reader an overview of the structure of the program, classes and methods not deemed necessary for this are not included in the diagram but can be seen in the source-code.

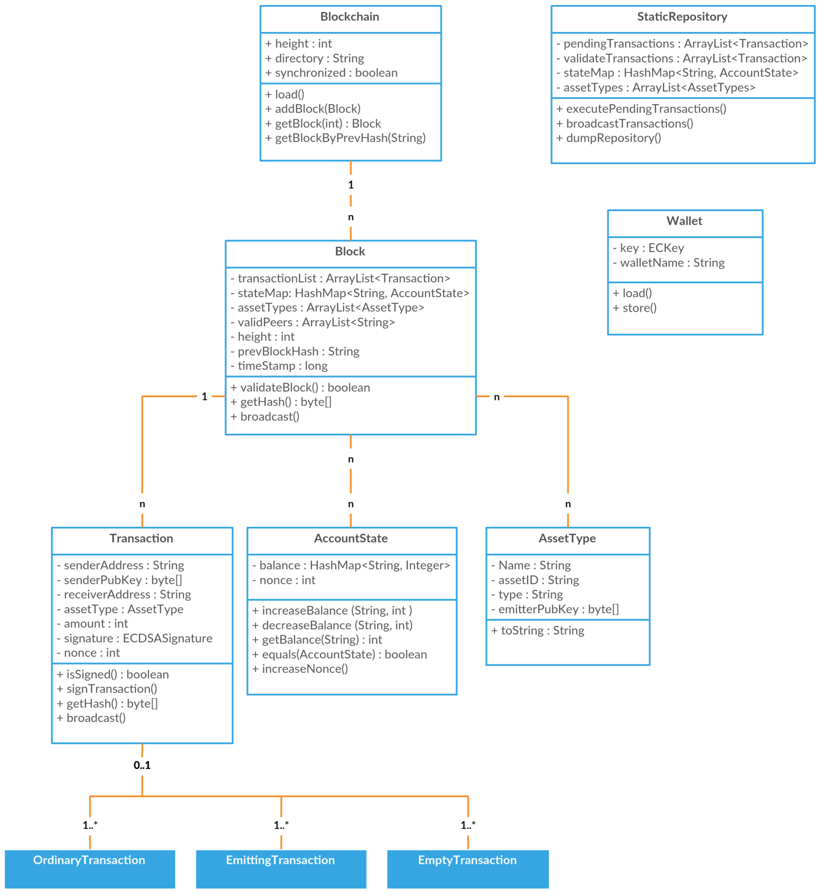

Figure 7: The UML class-diagram of some of the classes in the core package.

`AccountState` Represents the state of an account. Each account state has a map of asset types and balances as well as a 'nonce' used to ensure that a transaction can only be processed once.

`Transaction` The data structure used to conduct various types of transactions between accounts. Each transaction has a sender and a receiver address as well as the asset type and the amount to transfer. For validation purposes each transaction also contains the digital signature of the sender. To ensure that each transaction is only processed once each transaction has a nonce corresponding to the nonce of the senders account state at the time the transaction was created. The `Transaction` class is defined as an abstract class on which three subclasses extends.

`OrdinaryTransaction` Extends `Transaction`. Transacts an asset from one account to another.

`EmittingTransaction` Extends `Transaction`. Either creates a new asset type or issues more shares of an existing asset type.

`EmptyTransaction` Extends `Transaction`. Notifies the network that a new account has been created.

`StaticRepository` A container where transactions are validated and applied to the state. It has a list of pending transactions, i.e transactions that have been received but not yet processed, a list of validated transactions and a map of the current account states. When a node creates a new block the state and the validated transactions in the `StaticRepository` are put in the block.

Start The start package contains two classes, `Start` and `User`. The program is started by running the class `Start`, the user needs to provide an IP-address and a directory to store the local files in. The `User` class handles the different processes such as the server and miner, these processes are explained in more detail later in this section.

Crypto The crypto package contains classes used for cryptographic tasks such as generating private and public keys and hashing and signing messages. The code is originally from the java version of the open-source project Ethereum. Two important classes in the package are `HashUtil` and `ECKey`. `HashUtil` has methods for cryptographically hashing messages using the SHA-3 hash function and is utilized in the program for tasks such as hashing transactions and creating merkle hashes. `ECKey` is the class that represents the elliptic curve public and private key. The class provides various methods for generating new key pairs and for signing and verifying messages. Each user holds an ECKey object that contains the users private key and is used to digitally sign messages.

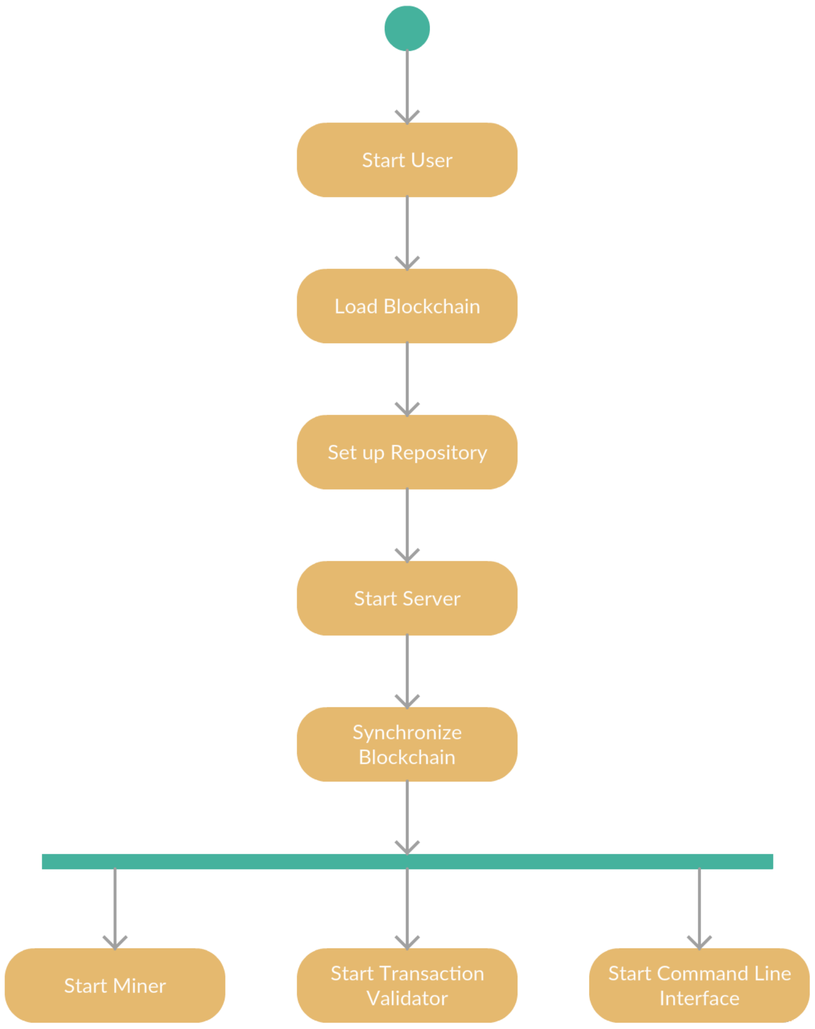

Figure 8: The UML activity diagram of the start-up process.

Network The network package contains three classes: `MultipleSocketServer`, `Client` and `Synchronize`. `MultipleSocketServer` is the server class and is used for receiving and handling incoming messages. The `Client` class is used to send messages, either to a specific node or to the entire network. `Synchronize` is run at the start-up to synchronize the local version of the blockchain with the network.

Methods and processes

Starting the program Figure 8 depicts a UML activity diagram of the process that is run each time the program is started. First the Blockchain is loaded using the load method in the Blockchain class. In this step the local copy of the blockchain is validated and the height is retrieved. In the next step the StaticRepository is set up using the state in the latest block of the blockchain. As this is done the server is started and the node starts to synchronize its local version of the Blockchain with the network. As the synchronization is completed the mining process, the transaction validation process and the command line interface are started in parallel.

Mining process The mining process is used to generate new blocks and is of a simple round robin type. Each node has a certain probability to create the next block. When a block is created it is broadcast to the network. Each node validates an incoming block in a number of ways, most notably by checking that the transactions in the block are valid and that the blocks previous block hash agrees with the nodes local version of the blockchain.

Transaction validation process The transaction validation process executes the pending transactions in the repository. The frequency with which the pending transactions are executed can be altered depending on the transaction volume to be handled, the default setting is one execution per second.

Command line interface The command line interface provides users a way to conduct operations such as creating new accounts and create and transact assets by typing in commands in the console. For a full description of the available commands the reader is referred the source code in appendix A.1.14.

Creating and broadcasting a transaction Figure 9 depicts the process of a node creating a new transaction. When creating a transaction the user needs to provide information about the receiver, the asset-type to transfer and the quantity to transfer. Before broadcasting the transaction to the network a validation process is run. In this process it is ensured that the sender has enough funds, that the signature is correct and that the transaction has not already been processed. Each transaction holds a list of the nodes that has seen the transaction. A node only re-broadcast a transaction to nodes that has not yet seen the transaction.

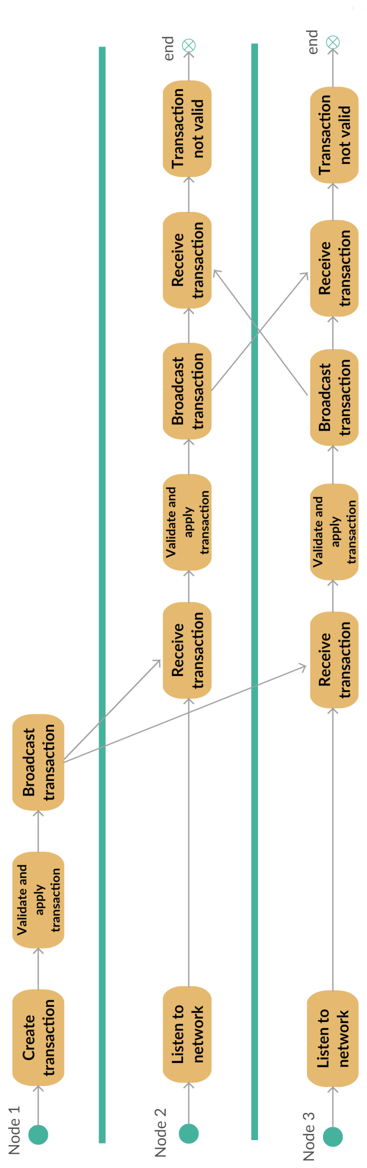

Figure 9: The process of creating and broadcasting a transaction in a network of three nodes.

Creating a new asset type To create a new asset-type the user needs to provide the necessary asset type information as well as the number of shares to issue. The shares are issued to the account of the creating node by an EmittingTransaction. This special type of transaction only differs from an OrdinaryTransaction in the way that the creator of the transaction does not need to have the transacted assets on its account to be considered valid.

7 Conclusions

From the interviews conducted with the actors on the Nordic capital market the following conclusions are drawn:

- In the Nordic region several actors are actively working with the development of DLTs.
- DLTs are likely to have an impact in several areas of the Nordic capital market. Most notably is the post trade area of clearing and settlement, but also free standing applications as well as internal processes are likely to be affected.
- Smart contracts are of particular interest for increasing straight through processing in many financial processes.
- For DLTs to be able to replace current post-trade systems the scaling in terms of transaction volume needs to be improved compared to current public ledgers.
- Permissioned ledgers are likely to be used if and when DLTs are implemented in the financial infrastructure.
- The process of standardization is likely to pose the greatest challenge for DLTs to be used within the Nordic capital market.

From the PoC the following conclusions are drawn:

- From a technical point of view creating an independent permissioned distributed ledger application does not require large resources. Assuming code for the public-key cryptography is given it is possible for one full-time resource to deliver a working application in three months.[14]

8 Discussion

Undoubtedly there is a large interest for DLTs within the financial industry in the Nordic region. Actors such as exchanges and banks have been committing increasingly amounts of resources to examine and develop DLT-based applications and PoCs the last few years. There are still technical challenges to be solved if DLTs are to be fully utilized with clearing and settlement in the capital market though. The technology would need to be able to handle vastly greater transaction volumes than is currently possible with, for example, the

[14]The application was created during the course of six months at roughly half-time pace.

Bitcoin-network and the transparency would need to be controlled to hinder leakage of sensitive information.

A plausible solution to the scaling problem is that of using a permissioned ledger together with a PBFT-based consensus process. PBFT-protocols have been the target of decades of research and have already proven their robustness and efficiency as a part of many file-sharing systems and are therefore reasonable to direct the most attention to when developing an efficient permissioned ledger consensus protocol. A permissioned ledger would also allow the users to control which nodes that have access to the blockchain, thereby solving at least one part of the transparency problem. The other part is that of the trusted nodes still having access to all transactions conducted over the network. This might arguably be the most troubling aspect for DLT's potential use in the capital market, since it would require banks and other institutions to give up information to each other that is today kept confidential. Finding solutions to this, either by adjusting the attitude for what information should be confidential and not or by developing technical solutions is a necessity should DLTs become the way to conduct clearing and settlement in the capital market.

It seems though that the technical difficulties of the technology are likely to be solved in the not too distant future. At the time of writing several large projects are ongoing where different ways of increasing the possible transaction volume and handling transparency are examined.[15] The main difficulty is likely to be in that of finding a common standard that is accepted by all involved actors, an achievement that would require co-operation among several independent institutions. Here Swedish banks have already proven their ability in form of the inter-bank instant payment service Swish. Seeing as Sweden and the Nordics in general also is an early adopter when it comes to digitalising financial services, it is not unlikely that DLTs would be rather quickly adopted.[16]

Beyond that of clearing and settlement there are still use-cases for DLTs on the capital market where for example the transparency problem would be less of an issue. For example, if banks were to employ internal distributed ledgers to handle their accounting smart-contracts could be used to increase straight trough processing in many processes. Another example is that of more independent financial applications such as the distributed ledger Linq created by Nasdaq. For these kind of use-cases existing technology is already sufficient and many new applications are likely to go public in the upcoming years.

As with many disruptive technologies it is hard to speculate in which areas DLTs will have the greatest impact. Twenty years ago few would have foreseen that the internet would play such a critical role in our society as it does today, and even fewer would have foreseen the endless use-cases that it provides. DLTs are currently in a phase of rapid development and much have happened only in the time that this thesis have been written. Regardless if DLTs as they look

[15]For example the Hyperledger project is developing permissioned blockchains using PBFT consensus with the specific aim of improving scaling. At the same time Ethereum is working on improving their public-ledger solution by using a variant of the PoS consensus protocol.

[16]See for example `http://www.reuters.com/article/nordic-cashless-idUSL6N0UM1TY20150109` [accessed 2016-06-23]

today would prove to be non-compliant with the capital market, the entrance of DLTs have triggered a major increase in effort put into improving post-trade systems. This effort might not lead to a direct implementation of DLTs into the capital market infrastructure, but in some form it is highly likely to lead to improvements of the systems as they look today.

References

[1] Wiebe Ruttenberg. *Distributed ledger technologies in securities post-trading.* European Central Bank Occasional Paper Series, p. 6.

[2] Accenture Capital Markets. *Blockchain in the Investment Bank.* 2015.

[3] Oliver Wyman Euroclear. *Blockchain in Capital Markets.* 2016.

[4] UK Government Chief Scientific Adviser. *Distributed Ledger Technology: beyond block chain.* 2016.

[5] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* 2009.

[6] Paolo Tasca. *Digital Currencies: Principles, Trends, Opportunities, and Risks.* 2015.

[7] Gavin Wood. *Etereum: A Secure Decentralized General Transaction Ledger.* 2014.

[8] Nasdaq. *Nasdaq Linq enables first-ever private security issuance documented with blockchain technology.* `http://ir.nasdaq.com/releasedetail.cfm?releaseid=948326`. [Online; Accessed 2016-06-22].

[9] Andreas M. Antonopoulos. *Mastering Bitcoin.* 1st. O'Reilly Media, Inc, 2014.

[10] William Stallings. *Cryptography and network security : principles and practice.* Boston: Prentice Hall, 2011. Chap. 11.3 - Requirements and security, pp. 327–356.

[11] William Stallings. *Cryptography and network security : principles and practice.* Boston: Prentice Hall, 2011.

[12] Whitfield Diffie and Martin E. Hellman. *New Directions in Cryptography.* 1976.

[13] Leonard Adleman Ron Rivest Adi Shamir. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.* 1978.

[14] William Stallings. *Cryptography and network security : principles and practice.* 5th. Boston: Prentice Hall, 2011. Chap. 9.1 - Principles of public-key cryptosystems, pp. 275–276.

[15] Neal Koblitz. "Elliptic curve cryptosystems". In: *Mathematics of Computation.* Vol. 48. 1987, pp. 203–209.

[16] Victor S. Miller. "Use of elliptic curves in cryptography". In: *Advances in Cryptology — CRYPTO '85 Proceedings.* 1985, pp. 417–426.

[17] Ralph C. Merkle. *Advances in Cryptology — CRYPTO '87: Proceedings.* Ed. by Carl Pomerance. Springer Berlin Heidelberg, 1988. Chap. A Digital Signature Based on a Conventional Encryption Function, pp. 369–378.

[18] Andreas M. Antonopoulos. *Mastering Bitcoin.* 1st. 2014, pp. 111–134.

[19] Vitalik Buterin. *Ethereum Whitepaper.* `https://github.com/ethereum/wiki/wiki/White-Paper`. [Online; accessed 2016-06-22].

[20] Vitalik Buterin. *On Public and Private Blockchains.* [Online; accessed 2016-06-22].

[21] Moni Naor Cynthia Dwork. *Pricing via Processing or Combatting Junk Mail.* 1993.

[22] Ari Jules Markus Jakobsson. *Proofs of Work and Bread Pudding Protocols.* 1999.

[23] Adam Back. *Hashcash - A Denial of Service Counter-Measure.* 2002.

[24] Christopher Malmo. *Bitcoin is Unsustainable.* `http://motherboard.vice.com/read/bitcoin-is-unsustainable`. [Online; accessed 2016-06-22].

[25] Bitcoinwiki. *Proof-of-Stake.* `https://en.bitcoin.it/wiki/Proof_of_Stake`. [Online; accessed 2016-06-22].

[26] Nxt. *Nxt Whitepaper.* `http://wiki.nxtcrypto.org/wiki/Whitepaper:Nxt`. [Online; accessed 2016-06-22].

[27] Scott Nadal Sunny King. *Peercoin Whitepaper.* `https://www.peercoin.net/whitepaper`. [Online; accessed 2016-06-22].

[28] Marshall Pease Leslie Lamport Robert Shostak. *The Byzantine Generals Problem.* Vol. 4. 1982, pp. 382–401.

[29] Barbara Liskov Miguel Castro. "Practical Byzantine Fault Tolerance". In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation.* 1999.

[30] Arthur Britto David Schwartz Noah Youngs. *The Ripple Protocol Consensus Algorithm.* 2014.

[31] Michael Simmons. *Securities Operations - A guide to Trade and Position Management.* John Wiley Sons, 2002.

[32] Rosalind Edwards. *What is qualitative interviewing.* Bloomsbury, 2013.

[33] Victoria Carlsson Lundström. "Impact from the blockchain technology on the Nordic capital market". MA thesis. Sweden: Uppsala universitet, 2016.

Interview references

[1] Johan Toll , *Nasdaq*, 2016-03-07, Stockholm, Personal interview.

[2] Anders Nyqvist, *SEB*, 2016-04-20, Stockholm, Personal interview.

[3] Anders Stenkrona, *Cryex*, 2016-04-05, Stockholm, Personal interview.

[4] Peter Strömberg , *Avanza*, 2016-02-26, Stockholm, Personal interview.

A Source code

A.1 Package: Core

A.1.1 AccountState

```java
package core;

import java.io.Serializable;
import java.util.HashMap;

/**
 * Represents the state of an account.  Holds a HashMap
 * containing asset-types and balances, and a 'nonce'
 * (a counter) used to hinder transactions from being
 * processed more than once.
 * @author Ludvig Backlund
 */

public class AccountState implements Serializable {
    /**
     * Auto generated serialVersionUID for serialization
     */
    private static final long serialVersionUID =
        -7946595253393611186L;

    /**
     *  HashMap containing the asset-type (different stocks for
         example)
     *  and the balance of each type.
     */
    HashMap<String, Integer> balance;

    /**
     *  Counter to ensure that a transaction only can be
         processed once.
     **/
    int nonce;

    public AccountState(){
        balance = new HashMap<>();
        nonce = 0;
    }

    /**
     * Increase the balance of an asset-type.
     * @param assetID - The asset-type to increase.
     * @param amount - The amount to increase with.
     */
     public void increaseBalance(String assetID, int amount){
         if(balance.containsKey(assetID)){
             balance.put(assetID, amount + balance.get(assetID))
                 ;
         } else
             balance.put(assetID, amount);
     }

     /**
      * Decrease the balance of an account
      * @param assetID - The asset-type to decrease.
      * @param amount - The amount to decrease with.
      */
     public void decreaseBalance(String assetID, int amount){
         if(balance.containsKey(assetID) && balance.get(assetID)
              >= amount){
             balance.put(assetID, balance.get(assetID) - amount)
                 ;
             if(balance.get(assetID) == 0)
                 balance.remove(assetID);
         }
         else
             System.err.println("Amount to transfer exceeds
                 balance.");
     }

     /**
      * To string method.
      * @return A string representation of the account.
      */
     public String toString(){
         return this.balance.toString();
     }

     /** Return the balance associated with an asset type
      * @param assetID - Asset type you want the balance for.
      * @return - The balance of the asset type.
      */
     public int getBalance(String assetID){
         if(balance.get(assetID) == null)
             return 0;
         else
             return balance.get(assetID);
     }

     /** Check if this AccountState is equal to another
         AccountState,
      *  (i.e. contains the same asset-types and balances.)
      * @param accountState - the accountState to check against.
      * @return true if equal, else false.
      */
     public boolean equals(AccountState accountState){
         for(String key: balance.keySet()){
             if(!balance.get(key).equals(accountState.getBalance
                 (key)))
                     return false;
         }
         return true;
     }

     /**
      * Increase the nonce (This is done for every
      * transaction sent from this AccountState).
      */
     public void increaseNonce(){
         nonce++;
     }

     /**
      * Get the nonce of this AccountState
      * @return nonce - the current nonce of this AccountState.
      */
     public Integer getNonce(){
         return nonce;
     }
 }
```

A.1.2 AssetType

```java
package core;
import java.io.Serializable;

import org.apache.commons.codec.binary.Hex;

/**
 * Class representing an asset type. Only the creator of the
     asset type
 * can issue more shares. This is achieved by having the public
     key of
 * the creator stored in the AssetType. To issue more shares a
     Transaction
 * signed with the corresponding private key must be created.
 * @author Ludvig Backlund
 */

public class AssetType implements Serializable {

    private static final long serialVersionUID =
        -4678916248790217977L;

    /**
     * Name of the asset type.
     */
    String name;

    /**
     * Unique identifier (ISIN-code etc.)
     */
    String ID;

    /**
     * Type of asset (i.e. stock, bond...)
     */
    String type;

    /**
     * Name of the emittent of this AssetType.
     */
    String emitterName;

    /**
```

```
     * The public key of the emittent of this AssetType
     */
    byte[] emitterPubKey;

    /**
     * Construct an AssetType deciding all fields.
     * @param name
     * @param ID
     * @param type
     * @param emitterName
     * @param emitterPubKey
     */
    public AssetType(String name, String ID, String type, String
         emitterName, byte[] emitterPubKey){
        this.name = name;
        this.ID = ID;
        this.type = type;
        this.emitterName = emitterName;
        this.emitterPubKey = emitterPubKey;
    }

    /**
     * Construct an empty AssetType.
     */
    public AssetType(){
        name = "";
        ID = "";
        type = "";
        emitterPubKey = new byte[0];
    }

    /**
     * toString method
     * @return A String representation of this AssetType.
     */
    public String toString(){
        return "name: " + name + "\r" + "ID: " + ID  + "\r" + "
            type: " + type + "\r" + "emitterPubKey: "
    + Hex.encodeHexString(emitterPubKey) + "\r" + "emitter name:
         " + emitterName;
    }

    /**
     * Get the ID.
     * @return The ID
     */
    public String getID(){
        return ID;
    }

    /**
     * Get the name of the AssetType.
     * @return
     */
    public String getName(){
        return name;
    }

    /**
     * Get the public key of the emittent of this asset type.
     * @return
     */
    public byte[] getPubKey(){
        return emitterPubKey;
    }

}
```

A.1.3 Block

```
package core;

import java.util.List;

import org.apache.commons.codec.DecoderException;
import org.apache.commons.codec.binary.Hex;


import crypto.HashUtil;
import exceptions.TransactionNotValidException;
import network.Client;

import java.io.Serializable;
import java.nio.ByteBuffer;
import java.util.ArrayList;
import java.util.HashMap;

/**
 * Class representing a block.
 * @author Ludvig Backlund
 */

public class Block implements Serializable{

    private static final long serialVersionUID = 1L;

    /**
     *  List of transactions.
     */
    protected List<Transaction> transactionList;

    /**
     *  HashMap of public keys and AccountStates
     */
    protected HashMap<String, AccountState> stateMap;

    /**
     *  List of all existing assetTypes
     */
    private ArrayList<AssetType> assetTypes;

    /**
     * List of the IP-addresses of the valid peers.
     */
    private List<String> validPeers = new ArrayList<>();

    /**
     * This Blocks number in the blockchain, genesis number is
         0.
     */
    private int height;

    /**
     * The SHA-3 hash of the preceeding block in the blockchain.
     */
    private String prevBlockHash;

    /**
     * The unix epoch time for the creation of this block.
     */
    protected long timeStamp;


    /**
     *  Construct a block containing an empty transactionlist,
         statemap and assetList.
     */
    public Block(){
        this.transactionList = new ArrayList<>();
        this.stateMap = new HashMap<>();
        this.assetTypes = new ArrayList<>();
    }
    /**
     *  Construct a block containing inserting all fields but
         timeStamp.
     */
    public Block(List<Transaction> transactionList, HashMap<
        String, AccountState> stateMap,
            ArrayList<AssetType> assetTypes, String
                prevBlockHash, List<String> validPeers, int
                height){
        this.transactionList = transactionList;
        this.stateMap = stateMap;
        this.assetTypes = assetTypes;
        this.prevBlockHash = prevBlockHash;
        this.validPeers = validPeers;
        this.height = height;
        this.setTimeStamp();
    }

    /**
     * Add a transaction to transaction list.
     * @param transaction to be added.
     */
    public void addTransaction(Transaction transaction){
        this.transactionList.add(transaction);
    }

    /**
     *  Add an AccountState to the stateMap.
     *  The address is converted to a String as byte[] does not
```

```
         work well with HashMap.
     *  @param byteAddress - The address of the account.
     *  @param accountState - The state related to the address.
     */
    public void addAccount(byte[] byteAddress, AccountState
        accountState){
        String stringAddress = Hex.encodeHexString(byteAddress);
        if(stateMap.containsKey(stringAddress)){
            System.out.println("Account already exists.");
            return;
        }
        stateMap.put(stringAddress, accountState);
    }

    /**
     *  Validate that is is valid to add this block to the end
         of the blockchain.
     *  Validation is done with respect to the transactions, the
          timeStamp, the block-height,
     *  and the block-hash.
     * @return true if block is valid else false.
     */
    public boolean validateBlock(){
        if(validateTransactions(Blockchain.getLatestBlock())){
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            if(this.getTimeStamp() <  System.currentTimeMillis()
                ){
                if(Blockchain.getHeight() == this.getHeight()-1)
                    {
                    // Check hash only if not at genesis block.
                    if(Blockchain.getHeight() > -1){
                        if(Blockchain.getBlock(this.getHeight()
                            -1).getHash().equals(this.
                            getPrevHash()))
                            return true;
                        else{
                            System.err.println("Previous block
                                hash not valid");
                            return false;
                        }
                    }
                    return true;
                }
            } else {
            System.err.println("Time-stamp unvalid");
            }
        } else {
            System.err.println("Transactions in block not valid"
                );
        }

        return false;
    }
```

```java

    /**
     * Validate the transactions in a proposed block.
     * The transactions in the block are applied on the state of
         the latest block.
     * If the resulting state is equal to the state of the
        proposed block the
     * transactions in the proposed block are considered valid.
     * @return true if valid else false.
     */
    public boolean validateTransactions(Block latestBlock){

        if(this.getTransactionList().isEmpty())
            return true;

        Repository validationRepository = new Repository(
            latestBlock);
        List<Transaction> claimedTransactions = this.
            transactionList;

        for(Transaction transaction: claimedTransactions){
            validationRepository.addPendingTransaction(
                transaction);
        }

        try {
            validationRepository.executePendingTransactions();
        } catch (TransactionNotValidException e) {
            System.out.println(e.getMessage());
        }

        // Check that states are equal:
        if(stateMap.keySet().equals(validationRepository.
            getStateMap().keySet())){
                for(String key : stateMap.keySet()){
                if(!stateMap.get(key).equals(
                    validationRepository.getStateMap().get(key))
                    ){
                    return false;
                }
            }
            return true;
        }
        return false;
    }

    /**
     * Get the hash of this Block (currently the hash is made up
     * of the transactions, the timeStamp and the previous block
         hash).
     * @return blockHash - the SHA-3 hash of this block.
     */
    public String getHash(){

        // Get byte arrays:
        List<Transaction> transactionList = this.
            getTransactionList();
        byte[] transactionHash = new byte[0];
        if(!transactionList.isEmpty()){
            transactionHash = new Merkle(transactionList).
                getHash();
        }
        byte[] timestampBytes = ByteBuffer.allocate(Long.SIZE).
            putLong(this.getTimeStamp()).array();
        byte[] prevHashBytes = new byte[0];
        try {
            prevHashBytes = Hex.decodeHex(prevBlockHash.
                toCharArray());
        } catch (DecoderException e) {
            e.printStackTrace();
        }

        // Concatenate the byte arrays:
        byte[] blockBytes = new byte[transactionHash.length +
            timestampBytes.length + prevHashBytes.length];
        System.arraycopy(transactionHash, 0, blockBytes, 0,
            transactionHash.length);
        System.arraycopy(timestampBytes, 0, blockBytes,
            transactionHash.length, timestampBytes.length);
        System.arraycopy(prevHashBytes, 0, blockBytes,
            transactionHash.length + timestampBytes.length,
            prevHashBytes.length);

        // Hash and convert to hex string:
        byte[] blockHashBytes = HashUtil.sha3(blockBytes);
        String blockHash = Hex.encodeHexString(blockHashBytes);
        return blockHash;
    }

    /**
     * Check if an address exists in the stateMap.
     * @param address - the address to check.
     * @return true if address exists, else false.
     */
    public boolean existAccount(String address){
        return stateMap.containsKey(address);
    }

    /**
     * Broadcast this Block to all peers.
     */
    public void broadcast(){
        List<String> receivers = new ArrayList<>();
        receivers.add("broadcast");
        Object[] objectsToSend = {receivers, this};
        Client.main(objectsToSend);
    }

    //TODO: Checks and some mechanism to add peers to list.
    /** Add a valid peer to the list of valid peers */
    public void addValidPeer(String peer){
        validPeers.add(peer);
    }

    /**
     * Print the stateMap.
     */
    public void printState(){
        for(String key : stateMap.keySet())
            System.out.println("Address: " + key + " Balance: "
                + stateMap.get(key).toString());
    }

    /**
     * Get the stateMap.
     * @return A HashMap<> of the accounts and their states.
     */
    public HashMap<String, AccountState> getState(){
        return stateMap;
    }

    /**
     * Get the transactionList.
     * @return the list of transactions.
     */
    public List<Transaction> getTransactionList(){
        return transactionList;
    }

    /**
     * Get the hash of the previous block.
     * @return prevBlockHash
     */

    public String getPrevHash(){
        return prevBlockHash;
    }

    /**
     * Get the height.
     * @return height
     */
    public int getHeight(){
        return height;
    }

    /**
     * Get time stamp.
     * @return
     */
    public long getTimeStamp(){
        return timeStamp;
    }

    /**
     * Set the time stamp of the block in unix epoch (
        Milliseconds since 1 January 1970)
     */
    public void setTimeStamp(){
        this.timeStamp =  System.currentTimeMillis();
    }

    /**
     * Get the list of valid peers.
     * @return validPeers
     */
    public List<String> getValidPeers(){
        return validPeers;
    }

    /**
     * Set previous block hash.
     * @param hash
     */
    public void setPrevBlockHash(String hash){
        this.prevBlockHash = hash;
    }

    /**
     * Set the Block height.
     * @param height
     */
    public void setHeight(int height){
        this.height = height;
    }

    /**
     * Get the assetTypes.
     * @return
     */
    public ArrayList<AssetType> getAssetTypes(){
        return assetTypes;
    }

}
```

A.1.4 Blockchain

```
package core;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.nio.file.InvalidPathException;
import java.nio.file.Paths;
import java.util.HashMap;

import java.io.EOFException;
import java.io.File;

/**
 *  The interface against the local copy of the blockchain.
 *  @author Ludvig Backlund
 */
public class Blockchain {

```

```java
    /**
     *  Number of blocks currently in this blockchain
     */
    private static int height;

    /**
     * The path to the directory where the blocks are stored.
     */
    private static String directory;

    /**
     * Map keeping track of which peers this node is
         synchronized against.
     */
    private static HashMap<String,Boolean> isSynchronized;

    /** Construct a Blockchain with a set height.
     * @param The number of blocks currently in the blockchain.
         Genesis height = 0.
     */
    public Blockchain(int height){
        Blockchain.height = height;
    }

    /**
     * Construct an empty Blockchain
     */
    public Blockchain(){
        isSynchronized = new HashMap<String,Boolean>();
    }

    /**
     * Add a block to the end of the Blockchain.
     * @param block - the Block to add.
     **/
    public static void addBlock(Block block){
        try
        {
            File file = new File(directory + block.getHeight());
            FileOutputStream fileOut = new FileOutputStream(file
                );
            ObjectOutputStream out = new ObjectOutputStream(
                fileOut);
            out.writeObject(block);
            out.close();
            fileOut.close();
            height = block.getHeight();
        } catch(IOException i)
          {
              i.printStackTrace();
          }
    }

    /**
     * Get the latest block from the blockchain.
     * @return the latest block, if no block is found null is
         returned.
     */
    public static Block getLatestBlock(){
    try{
        File file = new File(directory + height);
        FileInputStream fileIn = new FileInputStream(file);
        ObjectInputStream in = new ObjectInputStream(fileIn);
        try{
            Block currentBlock = (Block) in.readObject();
            in.close();
            fileIn.close();
            return currentBlock;
        }catch(EOFException e){
            in.close();
            fileIn.close();
        }
        }catch(Exception i)
        {
        }
        return null;
    }

    /**
     * Get a Block by the Blocks order in the blockchain
     * @return the Block of height blockHeight, if no block is
         found null is returned.
     */
    public static Block getBlock(int blockHeight){
        try{
            File file = new File(directory + blockHeight);
            FileInputStream fileIn = new FileInputStream(file);
            ObjectInputStream in = new ObjectInputStream(fileIn)
                ;
            Block currentBlock = (Block) in.readObject();
            in.close();
            fileIn.close();
            return currentBlock;

        }catch(Exception i)
        {
            i.printStackTrace();
        }
        return null;
    }

    /**
     *  Load the blockchain. Done at startup to ensure the local
          copy of the blockchain
     *  is valid and to retrieve the current blockchain height.
     *
     *  First checks if the Genesis block exists (should be
         named "0"),
     *  if so the Genesis block is validated. If it is valid the
          current blockchain-height
     *  is set to 0 and the rest of the blockchain is validated.
          If no Genesis block exists
     *  it is created.
     *
     */
    public static void load(){

        height = - 1;
        File block = new File(directory + 0);

        if(block.exists()){
            // First validate the Genesis block:
            Block genesis = getBlock(0);

            if(!((Genesis) genesis).validateGenesis()){
                System.err.println("Gensis block invalid.");
                return;
            }
            height = 0;
            block = new File(directory + (height + 1));
            // Then validate the rest of the blockchain:
            while(block.exists()){
                Block currentBlock = getBlock(height + 1);
                if(!currentBlock.validateBlock()){
                    System.err.println("Block " + (height + 1) +
                         " invalid. Blockchain not valid.");
                    return;
                }
                height++;
                block = new File(directory + (height + 1));
            }
        } else{
            new Genesis();
            load();
        }
    }

    /**
     * Check that the directory provided by the user is valid.
     * @param directory - the directory path
     * @return true if directory path is valid else false
     */
    private static boolean checkDirectory(String directory){
        try{
            Paths.get(directory);
        } catch(InvalidPathException | NullPointerException e){
            return false;
        }
        return true;
    }

    /**
     * Get the current height (i.e. number of blocks-1) of the
        blockchain.
     * @return the current height of the blockchain.
     */
    public static int getHeight(){
        return height;
    }

    /** Set the directory where to store the blockchain
```

```
     *  Checks that the file path is valid. If so checks
     *  if directory exists, else it is created.
     *  @param directory - the absolute or relative path of the
         directory.
     **/
    public static void setDirectory(String directory){

        if(checkDirectory(directory)){
            File file = new File(directory);
            if(!file.exists())
                file.mkdirs();
            Blockchain.directory = directory;
        }

        }
    /**
     *  Get the directory path.
     *  @return The path of the directory.
     */
    public static String getDirectory(){
        return directory;
    }

    /**
     * Check whether this Blockchain is synchronized with a
         specific peer.
     * @param address - IP-address of the peer to check against.
     * @return true if synchronized else false.
     */
    public static boolean getIsSynchronized(String address){
        if(isSynchronized.containsKey(address))
            return isSynchronized.get(address);
        else
            return false;
    }

    /**
     *  Get a block by its previous block hash.
     *  @param hash - the prevBlockHash of the Block.
     *  @return The block that has hash as its prevBlockHash. If
          not found null is returned.
     **/
    public static Block getBlockByPrevHash(String hash){
        Block block;
        for(int i = 0; i < height+1; i++){
            block = Blockchain.getBlock(i);
            if(block.getPrevHash().equals(hash)){
                                return block;
            }
        }
            return null;
    }

    /**
     * Set if a peer is synchronized or not.
     * @param address - the IP-address of the peer.
     * @param value - true if synchronized else false.
     */
    public static void setSynchronized(String address, boolean
        value){
        isSynchronized.put(address, value);
    }

    /**
     * Set the height of the Blockchain.
     * @param height
     */
    public static void setHeight(int height){
        Blockchain.height = height;
    }
}
```

A.1.5 Transaction

```
package core;

import crypto.*;
import crypto.ECKey.ECDSASignature;
import network.Client;
import network.MultipleSocketServer;

import org.apache.commons.codec.binary.Hex;
import java.io.Serializable;
import java.nio.*;
import java.util.ArrayList;
import java.util.List;

/**
 * Abstract class for all Transactions.
 *
 * @author Ludvig Backlund
 */

public class Transaction implements Serializable {

    private static final long serialVersionUID =
        -758359160527238834L;

    /**
     *  Senders address (public key) as Hex String.
     */
    protected String senderAddress;

    /**
     * Senders address (public key) as byte[].
     */
    protected byte[] senderPubKey;

    /**
     * Receivers address (public key) as Hex String.
     */
    protected String receiverAddress;

    /**
```

```java
     * Asset-type to transfer.
     */
    protected AssetType type;

    /**
     * The amount to transfer
     */
    protected Integer amount;

    /**
     * Time-stamp (Linux epoch time)
     */
    protected int timeStamp;

    /**
     * The hash of the transaction data.
     */
    protected byte[] messageHash;

    /**
     * The senders ECDSASignature.
     */
    protected ECDSASignature signature;

    /**
     * IP-address of the nodes that sent the transaction.
     **/
    protected List<String> senderIP = new ArrayList<>();

    /**
     *  Flag telling if the transaction is signed or not.
     */
    protected boolean isSigned = false;

    /**
     * Counter to ensure that a transaction can only be
         processed once
     **/
    protected int nonce;

    /**
     * The ID of an asset used with ordinary transactions
     */
    protected String assetID;

    /** Constructs an unsigned transaction
     *
     * @param sender - Address of the sender.
     * @param receiver - Address of the receiver.
     * @param type - Asset type to transact.
     * @param amount - Amount to transact.
     */
    public Transaction(byte[] senderAddress, byte[]
        receiverAddress, AssetType type, int amount){
        this.senderAddress = Hex.encodeHexString(senderAddress);
        this.receiverAddress = Hex.encodeHexString(
            receiverAddress);;
        this.amount = amount;
        this.type = type;
        this.assetID = type.getID();
        this.nonce = StaticRepository.getStateMap().get(this.
            senderAddress).getNonce();
        setTimeStamp();
    }


    /** Constructs a signed transaction (Only used locally in
        signTransaction method)
     *
     * @param sender - Address of the sender.
     * @param receiver - Address of the receiver.
     * @param type - Asset type to transact.
     * @param amount - Amount to transact.
     * @param signature - The senders signature.
     * @param messagehash - SHA-3 hash of the transaction. Used
         in verification.
     */
    protected Transaction(String senderAddress, byte[]
        senderPubKey, String receiverAddress, AssetType type,
            Integer amount, ECDSASignature signature, int
                timeStamp, byte[] messageHash, int nonce){
        this.senderAddress = senderAddress;
        this.senderPubKey = senderPubKey;
        this.receiverAddress = receiverAddress;
        this.amount = amount;
        this.type = type;
        this.assetID = type.getID();
        this.signature = signature;
        this.messageHash = messageHash;
        this.timeStamp = timeStamp;
        this.isSigned = true;
        this.nonce = nonce;
    }

    /** Constructs an unsigned transaction
     *
     * @param sender - Address of the sender.
     * @param receiver - Address of the receiver.
     * @param type - Asset type to transact.
     * @param amount - Amount to transact.
     */
    public Transaction(byte[] senderAddress, byte[]
        receiverAddress, String assetID, int amount){
        this.senderAddress = Hex.encodeHexString(senderAddress);
        this.receiverAddress = Hex.encodeHexString(
            receiverAddress);;
        this.amount = amount;
        this.assetID = assetID;
        this.nonce = StaticRepository.getStateMap().get(this.
            senderAddress).getNonce();
        setTimeStamp();
    }

    /** Constructs a signed transaction (Only used locally in
         signTransaction method)
     *
     * @param sender - Address of the sender.
     * @param receiver - Address of the receiver.
     * @param type - Asset type to transact.
     * @param amount - Amount to transact.
     * @param signature - The senders signature.
     * @param messagehash - SHA-3 hash of the transaction. Used
         in verification.
     */
    protected Transaction(String senderAddress, byte[]
        senderPubKey, String receiverAddress, String assetID,
            Integer amount, ECDSASignature signature, int
                timeStamp, byte[] messageHash, int nonce){
        this.senderAddress = senderAddress;
        this.senderPubKey = senderPubKey;
        this.receiverAddress = receiverAddress;
        this.amount = amount;
        this.assetID = assetID;
        this.signature = signature;
        this.messageHash = messageHash;
        this.timeStamp = timeStamp;
        this.isSigned = true;
        this.nonce = nonce;
    }

    /**
     * Get the SHA-3 hash of this transaction.
     * @return the hash.
     */

    public byte[] getHash(){
        // Get a byte array to do the hashing on from the
            transaction data:
        byte[] senderBytes = senderAddress.getBytes();
        byte[] receiverBytes = receiverAddress.getBytes();
        byte[] amountBytes = ByteBuffer.allocate(Integer.SIZE).
            putInt(amount).array();
        byte[] typeBytes = assetID.getBytes();
        byte[] timeStampBytes = ByteBuffer.allocate(Integer.SIZE
            ).putInt(timeStamp).array();
        byte[] nonceBytes = ByteBuffer.allocate(Integer.SIZE).
            putInt(nonce).array();
        byte[] messageBytes = new byte[senderBytes.length +
            receiverBytes.length + amountBytes.length +
                                        typeBytes.length +
                                            timeStampBytes.length
                                             + nonceBytes.length
                                            ];
        System.arraycopy(senderBytes, 0, messageBytes, 0,
            senderBytes.length);
        System.arraycopy(receiverBytes, 0, messageBytes,
            senderBytes.length, receiverBytes.length);
        System.arraycopy(amountBytes, 0, messageBytes,
            senderBytes.length + receiverBytes.length,
            amountBytes.length);
        System.arraycopy(typeBytes, 0, messageBytes, senderBytes
```

```java
            .length + receiverBytes.length + amountBytes.length,
             typeBytes.length);
        System.arraycopy(timeStampBytes, 0, messageBytes,
            senderBytes.length + receiverBytes.length +
            amountBytes.length + typeBytes.length,
            timeStampBytes.length);
        System.arraycopy(nonceBytes, 0, messageBytes,
            senderBytes.length + receiverBytes.length +
            amountBytes.length + typeBytes.length +
            timeStampBytes.length, nonceBytes.length);

        // Hash the messageBytes:
        byte[] messageHash = HashUtil.sha3(messageBytes);
        return messageHash;
    }

    /**
     * Broadcast this transaction to all peers.
     */
    public void broadcast(){
        List<String> receivers = new ArrayList<>();
        receivers.add("broadcast");
        this.senderIP.add(MultipleSocketServer.getIP());
        Object[] objectsToSend = {receivers, this};
        Client.main(objectsToSend);
    }

    /**
     * Set the time stamp of the transaction in unix epoch (
         Seconds since 1 January 1970).
     */
    public void setTimeStamp(){
        this.timeStamp = (int) (System.currentTimeMillis() /
            1000L);
    }

    /**
     * Add an IP-address to senderIP.
     * @param senderIP
     */
    public void setSenderIP(String senderIP){
        this.senderIP.add(senderIP);
    }

    /**
     * Get the senderIP list.
     * @return
     */
    public List<String> getSenderIP(){
        return this.senderIP;
    }

    /**
     * Get the nonce.
     * @return
     */
    public int getNonce(){
        return this.nonce;
    }

    /**
     *  Get the senders address
     * @return the Hex String representation of the senders
         address
     */
    public String getSenderAddress(){
        return senderAddress;
    }

    /**
     * Get the senders public key
     * @return the byte[] representation of the senders address.
     */
    public byte[] getSenderPubKey(){
        return senderPubKey;
    }

    /**
     * Get the receivers address.
     * @return The Hex String representation of the receivers
         address.
     */
    public String getReceiverAddress(){
        return receiverAddress;
    }

    /**
     * Get the amount.
     * @return
     */
    public int getAmount(){
        return amount;
    }

    /**
     * Get the asset type.
     * @return
     */
    public AssetType getType(){
        return type;
    }

    /**
     * Get the hash of the transaction data.
     * @return
     */
    public byte[] getMessageHash(){
        return messageHash;
    }

    /**
     *  Get the ECDSASignature.
     *  @return
     */
```

```java
    public ECDSASignature getSignature(){
        return signature;
    }

    /**
     * Check if the transaction is signed or not.
     * @return true if signed else false.
     */
    public boolean isSigned(){
        return isSigned;
    }

    /**
     * Get assetID
     * @return assetID
     */
    public String getAssetID(){
        return assetID;
    }
}
```

A.1.6 OrdinaryTransaction

```java
package core;

import crypto.ECKey;
import crypto.ECKey.ECDSASignature;

/**
 * Class representing an "ordinary" transaction, i.e. a
     transaction that transfers an asset
 * from one account to another.
 * @author Ludvig Backlund
 */

public class OrdinaryTransaction extends Transaction {

    private static final long serialVersionUID =
        966032047732505698L;

    /**
     * Create an unsigned OrdinaryTransaction.
     * @param senderAddress
     * @param receiverAddress
     * @param assetID
     * @param amount
     */
    public OrdinaryTransaction(byte[] senderAddress, byte[]
        receiverAddress, String assetID, int amount) {
        super(senderAddress, receiverAddress, assetID, amount);
    }

    /**
     * Create a signed OrdinaryTransaction.
     * @param senderAddress
     * @param pubKey
     * @param receiverAddress
     * @param assetID
     * @param amount
     * @param signature
     * @param timeStamp
     * @param messageHash
     * @param nonce
     */
    public OrdinaryTransaction(String senderAddress, byte[]
        pubKey, String receiverAddress, String assetID,
            Integer amount, ECDSASignature signature, int
                timeStamp, byte[] messageHash, int nonce) {
        super(senderAddress, pubKey, receiverAddress, assetID,
            amount, signature, timeStamp, messageHash, nonce);
    }

    /**
     * Sign this transaction. An ECKey is used to sign a byte
         array derived from the input transactions fields.
     * @param key - The ECKey signing the transaction.
     * @return A signed transaction
     */
    public OrdinaryTransaction signTransaction(ECKey key){
        byte[] messageHash = this.getHash();
        signature = key.doSign(messageHash);
        return new OrdinaryTransaction(senderAddress, key.
            getPubKey(), receiverAddress, assetID, amount,
            signature, timeStamp, messageHash, nonce);
    }
}
```

A.1.7 EmittingTransaction

```java
package core;

import crypto.ECKey;
import crypto.ECKey.ECDSASignature;

/**
 * Transaction for either creating a new AssetType or for
     issuing
 * more shares of an existing AssetType.
 * @author Ludvig Backlund
 */

public class EmittingTransaction extends Transaction {

    private static final long serialVersionUID =
        -3304799141746904602L;

    /**
     * Construct a new unsigned EmittingTransaction.
     * @param senderAddress
     * @param type
     * @param amount
     */
    public EmittingTransaction(byte[] senderAddress, AssetType
        type, int amount) {
```

```
        super(senderAddress, senderAddress, type, amount);
    }

    /**
     * Construct a new signed EmittingTransaction.
     * @param senderAddress
     * @param pubKey
     * @param receiverAddress
     * @param type
     * @param amount
     * @param signature
     * @param timeStamp
     * @param messageHash
     * @param nonce
     */
    public EmittingTransaction(String senderAddress, byte[]
        pubKey, String receiverAddress, AssetType type,
            Integer amount, ECDSASignature signature, int
                timeStamp, byte[] messageHash, int nonce) {
        super(senderAddress, pubKey, receiverAddress, type,
            amount, signature, timeStamp, messageHash, nonce);
    }

    /**
     * Sign this transaction. An ECKey is used to sign a byte
         array derived from the input transactions fields.
     * @param key - The ECKey signing the transaction.
     * @return A signed transaction
     */
    public EmittingTransaction signTransaction(ECKey key){
        byte[] messageHash = this.getHash();
        signature = key.doSign(messageHash);
        return new EmittingTransaction(senderAddress, key.
            getPubKey(), receiverAddress, type, amount,
            signature, timeStamp, messageHash, nonce);
    }
}
```

A.1.8 EmptyTransaction

```
package core;

import crypto.ECKey;
import crypto.ECKey.ECDSASignature;

/**
 * Transaction used for telling the network that we created a
     new
 * account.
 * @author Ludvig Backlund
 *
 */

public class EmptyTransaction extends Transaction{

    private static final long serialVersionUID =
        1254158150455163558L;
```

```java

    /**
     * Construct a new unsigned EmptyTransaction.
     * @param senderAddress
     */
    public EmptyTransaction(byte[] senderAddress) {
        super(senderAddress, senderAddress, "", 0);
    }

    /**
     * Construct a new Signed EmptyTransaction.
     * @param senderAddress
     * @param pubKey
     * @param receiverAddress
     * @param assetID
     * @param amount
     * @param signature
     * @param timeStamp
     * @param messageHash
     * @param nonce
     */
    public EmptyTransaction(String senderAddress, byte[] pubKey,
         String receiverAddress, String assetID,
            Integer amount, ECDSASignature signature, int
                timeStamp, byte[] messageHash, int nonce) {
        super(senderAddress, pubKey, receiverAddress, assetID,
            amount, signature, timeStamp, messageHash, nonce);
    }

    /**
     * Sign this transaction. An ECKey is used to sign a byte
         array derived from the input transactions fields.
     * @param key - The ECKey signing the transaction.
     * @return A signed transaction
     */
    public EmptyTransaction signTransaction(ECKey key){
        byte[] messageHash = this.getHash();
        signature = key.doSign(messageHash);
        return new EmptyTransaction(senderAddress, key.getPubKey
            (), receiverAddress, assetID, amount, signature,
            timeStamp, messageHash, nonce);
    }
}
```

A.1.9 Genesis

```java
package core;

import java.util.ArrayList;
import java.util.HashMap;

/**
 * Represents the Genesis block. When a node is starting an
     instance
 * of the program with no local version of the blockchain stored
 * the Genesis block is created. All nodes participating on the
     same
 * blockchain must have the same version of the Genesis block.
 *
 * @author Ludvig Backlund
 *
 */
public class Genesis extends Block {

    private static final long serialVersionUID = 1L;

    /**
     * Construct a new Genesis block.
     */
    public Genesis(){
        this.timeStamp = 0;
        this.setPrevBlockHash("");
        this.transactionList = new ArrayList<>();
        this.stateMap = new HashMap<>();
        this.setHeight(0);
        this.addValidPeer("127.0.0.1");
        this.addValidPeer("127.0.0.2");
        this.addValidPeer("127.0.0.3");
        this.addValidPeer("127.0.0.4");
        Blockchain.addBlock(this);

        System.out.println("Genesis block created.");
    }

    /**
     * Validate the genesis block by checking that total height
     * of blockchain == -1, hash of previous block equals "",
     * that no transactions are contained and timestamp == 0.
     * @return true if valid else false.
     */
    public boolean validateGenesis(){
        if(this.getHeight() == 0 && this.getPrevHash().equals(""
            )
                && this.getTransactionList().isEmpty() && this.
                    getTimeStamp() == 0){
            return true;
        }
        return false;
    }
}
```

A.1.10 Merkle

```java
package core;

import java.util.ArrayList;
import java.util.List;

import crypto.HashUtil;

/**
 * Class used to get the Merkle root hash from a list of
    Transactions.
 * @author Ludvig Backlund
 */

public class Merkle {

    /**
     * The Merkle root hash.
     */
    private byte[] rootHash;

    /**
     * Construct a Merkle-object containing the root hash of a set of transactions.
     * @param transactionList - The transactions to create the root has from.
     */
    public Merkle(List<Transaction> transactionList){
        List<byte[]> transactionHashes = toBytes(transactionList);
        this.rootHash = merkleHash(transactionHashes);
    }

    /**
     * Get a roothash from a list of transaction hashes.
     * @param transactionHashList - The list of transaction hashes.
     * @return the root hash of the transactionHashList.
     */
    private byte[] merkleHash(List<byte[]> transactionHashList){
        byte[] trans1;
        byte[] trans2;
        byte[] transConcatenated;
        List<byte[]> tempTransactionHashList = new ArrayList<>();
        // If even number of transactions
        if(transactionHashList.size() > 1 && transactionHashList.size() % 2 == 0){
            // Concatenate every two hashes
            for(int i = 0; i < transactionHashList.size(); i=i+2){
                trans1 = transactionHashList.get(i);
                trans2 = transactionHashList.get(i+1);
                transConcatenated = new byte[trans1.length + trans2.length];
                System.arraycopy(trans1, 0, transConcatenated, 0, trans1.length);
                System.arraycopy(trans2, 0, transConcatenated, trans1.length, trans2.length);
                tempTransactionHashList.add(HashUtil.sha3(transConcatenated));
            }
            return merkleHash(tempTransactionHashList);
        }
        else if(transactionHashList.size() > 1 && transactionHashList.size() % 2 != 0) {
            // Copy last hash and add it to the end of the list to get it even:
            transactionHashList.add(transactionHashList.get(
                transactionHashList.size()-1));
            return merkleHash(transactionHashList);
        }
        return transactionHashList.get(0);
    }

    /**
     * Convert a list of transaction to a list of the
        corresponding SHA-3 hashes.
     * @param transactionList - A list of the transactions to
        convert.
     * @return A list of the corresponding SHA-3 hashes.
     */
    private List<byte[]> toBytes(List<Transaction>
        transactionList){
        List<byte[]> transactionsHashList = new ArrayList<>();
        for(Transaction transaction: transactionList){
            transactionsHashList.add(transaction.getHash());
        }
        return transactionsHashList;
    }

    /**
     * Get the rootHash.
     * @return
     */
    public byte[] getHash(){
        return rootHash;
    }

}
```

A.1.11 Miner

```
package core;
/**
 * The process creating a new Block. For now a 'round robin'
    type is used
 * where each node has a certain likelihood to create a new
    block.
 *
 * @author Ludvig Backlund
 */
import java.util.Random;

public class Miner {

    /** Set the time interval here (in seconds)*/
    private static int minimumTime = 100;
    private static int maximumTime = 100;

    public static void main(){

        Random randomGenerator = new Random();

        while(true){
            try {
                Thread.sleep(randomGenerator.nextInt(maximumTime
                    *1000)+minimumTime*1000);
            } catch (InterruptedException e) {
            e.printStackTrace();
            }

            System.out.println("Mined new block!");
            StaticRepository.dumpRepository();
            new StaticRepository();
            Blockchain.getLatestBlock().broadcast();

        }

    }

}
```

A.1.12 StaticRepository

```
package core;

import java.util.List;

import org.apache.commons.codec.DecoderException;
import org.apache.commons.codec.binary.Hex;

import java.rmi.AlreadyBoundException;
import java.util.ArrayList;
import java.util.ConcurrentModificationException;
import java.util.HashMap;

import crypto.ECKey;
import exceptions.TransactionNotValidException;
import network.Client;

/**
 * Repository to keep the current state. Contains methods to
 * execute and validate transactions. As a new block
 * is created the current state of the StaticRepository is saved
      into a new
 * block. If a new Block is received the state of that block is
     loaded
 * into the StaticRepository. Validated Transactions that are
     not in the received
 * block are kept.
 *
 * @author Ludvig Backlund
 *
 */

public final class StaticRepository {

    /**
     *  Map of public keys (addresses) and AccountStates.
     */
    private static HashMap<String, AccountState> stateMap = new
        HashMap<>();
```

```java

    /**
     *  List of pending transactions
     */
    private static ArrayList<Transaction> pendingTransactions =
        new ArrayList<>();

    /**
     *  List of validated transactions
     */
    private static ArrayList<Transaction> validatedTransactions
        = new ArrayList<>();

    /**
     * List of all existing AssetTypes
     */
    private static ArrayList<AssetType> assetTypes = new
        ArrayList<>();

    /**
     *  Latest Block currently in blockchain.
     */
    private static Block latestBlock;

    /**
     *  Construct a repository from a Block.
     *  @param latestBlock
     */
    public StaticRepository(Block block){
        StaticRepository.latestBlock = block;
        StaticRepository.setStateMap(block.getState());
        StaticRepository.pendingTransactions = new ArrayList<>()
            ;
        StaticRepository.validatedTransactions = new ArrayList
            <>();
    }

    /**
     *  Construct a repository from the latest Block.
     *  @param latestBlock
     */
    public StaticRepository(){
        StaticRepository.latestBlock = Blockchain.getLatestBlock
            ();
        StaticRepository.stateMap = latestBlock.getState();
        StaticRepository.assetTypes = latestBlock.getAssetTypes
            ();
        StaticRepository.pendingTransactions = new ArrayList<>()
            ;
        StaticRepository.validatedTransactions= new ArrayList
            <>();
    }

    /**
     *  Construct a repository where some pending transactions
         are
     *  kept from the previous repository.
     *  @param pendingTransactions
     */
    public StaticRepository(ArrayList<Transaction>
        pendingTransactions){
        try{
            StaticRepository.latestBlock = Blockchain.
                getLatestBlock();

            /* Keep transactions from last Repository */
            HashMap<String, AccountState> mergedMap = new
                HashMap<>();
            mergedMap.putAll(latestBlock.getState());
            mergedMap.putAll(StaticRepository.getStateMap());
            StaticRepository.stateMap = mergedMap;
            StaticRepository.assetTypes = latestBlock.
                getAssetTypes();

            StaticRepository.pendingTransactions =
                pendingTransactions;
            StaticRepository.validatedTransactions = new
                ArrayList<>();
        } catch(Exception e){

        }
        }

    //TODO: Dependent transactions.
    /**
     *  Execute all transactions currently in pending
         transactions list.
     *  If receiver address does not exist in state it is
         created, if
     *  sender address does not exist in state we return.
     *  Adds successful transaction to the validated
         transactions list
     *  Non successful transactions are discarded.
     * @throws DecoderException
     *
     */
public static void executePendingTransactions() throws
    TransactionNotValidException {
    try{
        for(Transaction pendingTransaction: pendingTransactions)
            {
            String sender = pendingTransaction.getSenderAddress
                ();
            String receiver = pendingTransaction.
                getReceiverAddress();

            /** Check that transaction is signed **/
            try
            {
                if(!ECKey.verify(pendingTransaction.getHash(),
                    pendingTransaction.getSignature(),
                        pendingTransaction.getSenderPubKey())){
                    pendingTransactions = new ArrayList<>();
                    throw new TransactionNotValidException("
```

```java
                    Signature of sender not valid: " +
                    sender);
                }
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }

            /** If EmptyTransaction we are done. */
            if(pendingTransaction instanceof EmptyTransaction){
                //System.out.println("INSIDE EMPTY TRANSACTION")
                    ;
                stateMap.put(sender, new AccountState());
                getStateMap().get(sender).increaseNonce();
                validatedTransactions.add(pendingTransaction);
                pendingTransaction.broadcast();
                continue;
            }

            /** Check that transaction has not already been
                processed */
            if(!(stateMap.get(pendingTransaction.
                getSenderAddress()).getNonce() ==
                pendingTransaction.getNonce())){
                pendingTransactions = new ArrayList<>();
                throw new TransactionNotValidException("
                    Transaction nonce is not equal to Account
                    nonce" +
                        stateMap.get(pendingTransaction.
                            getSenderAddress()).getNonce() +
                            pendingTransaction.getNonce());
            }

            /** Check that both sender and receiver exist in
                state */
            if(!(getStateMap().containsKey(pendingTransaction.
                getSenderAddress())
                    && getStateMap().containsKey(
                        pendingTransaction.getReceiverAddress())
                        ))){
                pendingTransactions = new ArrayList<>();
                throw new TransactionNotValidException("Sender
                    or Receiver does not exist in state");
            }


            /** If EmittingTransaction no more validation is
                needed */
            if(pendingTransaction instanceof EmittingTransaction
                ){
                AssetType assetType = pendingTransaction.getType
                    ();
                /** Only the owner of the assetType can alter it
                    : */
                for(AssetType assetTypeTemp: assetTypes){
                    if(assetTypeTemp.getID().equals(assetType.
                        getID())){
                        if(!ECKey.verify(pendingTransaction.
                            getMessageHash(), pendingTransaction
                            .getSignature(),
                            assetTypeTemp.getPubKey())){
                            pendingTransactions = new ArrayList
                                <>();
                            throw new
                                TransactionNotValidException("
                                Signature does not match owner
                                of AssetType: " + assetType.
                                getName());
                        }
                    break;
                    }
                }
                assetTypes.add(assetType);
                stateMap.get(receiver).increaseBalance(
                    pendingTransaction.getAssetID(),
                    pendingTransaction.getAmount());
                stateMap.get(sender).increaseNonce();
                validatedTransactions.add(pendingTransaction);
                pendingTransaction.broadcast();
                continue;
            }

            /** Check that sender has enough balance on account
                */
            if(stateMap.get(sender).getBalance(
                pendingTransaction.getAssetID()) <
                pendingTransaction.getAmount()){
                throw new TransactionNotValidException("Not
                    enough balance on sender account: " + sender
                    );
            }

            /** Transaction is ordinary transaction */
            stateMap.get(receiver).increaseBalance(
                pendingTransaction.getAssetID(),
                pendingTransaction.getAmount());
            stateMap.get(sender).decreaseBalance(
                pendingTransaction.getAssetID(),
                pendingTransaction.getAmount());
            stateMap.get(sender).increaseNonce();
            validatedTransactions.add(pendingTransaction);
            pendingTransaction.broadcast();

        }
        pendingTransactions = new ArrayList<>();
    }
    catch(ConcurrentModificationException e){

    }
    }

    /**
     * Get Transactions that are uncommon in StaticRepository
```

```
         and a Block.
     * (used to keep transactions when receiving a new Block)
     * @param block - the block to compare against.
     * @return A list of the uncommon transactions.
     */
    public static ArrayList<Transaction> getUncommonTransactions
        (Block block){
        /** Get transactions from block and initialize a new
            transaction list */
        List<Transaction> blockTransactions = block.
            getTransactionList();
        ArrayList<Transaction> uncommonTransactions = new
            ArrayList<>();

        /** Add transactions that does not exist in the blocks
         *  transactions to new transaction-list.
         */
        boolean isCommon = false;
        for(Transaction myTransaction: validatedTransactions){
            for(Transaction blockTransaction: blockTransactions)
                {
                if(Hex.encodeHexString(myTransaction.
                    getMessageHash()).equals(Hex.encodeHexString
                    (blockTransaction.getMessageHash()))){
                    isCommon = true;
                    break;
                }
            }
            if(!isCommon){
                uncommonTransactions.add(myTransaction);
            }
        }

        return uncommonTransactions;

    }

    /**
     *  Dump the current repository into a new Block.
     */
    public static void dumpRepository(){
        Block block = new Block(validatedTransactions, stateMap,
             assetTypes,
                latestBlock.getHash(), latestBlock.getValidPeers
                    (), latestBlock.getHeight()+1);

        if(block.validateBlock()){
            Blockchain.addBlock(block);
            System.out.println("Block saved as: Block " + block.
                getHeight() + ", hash: " + block.getHash() +
                    ", time-stamp: " + block.getTimeStamp() + ",
                         nr of transactions: " + block.
                        getTransactionList().size());
        }
        else
            System.err.println("Could not add block to
                blockchain, block not valid.");
```

```java
    }

    /**
     * Broadcast validated transactions to all peers.
     */
    public static void broadcastTransactions(){
        for(Transaction transaction: validatedTransactions){
            /** Broadcast to everyone */
            List<String> receivers = new ArrayList<>();
            receivers.add("broadcast");
            Object[] objectsToSend = {receivers, transaction};
            Client.main(objectsToSend);
        }

    }

    /**
     * Print the current account state.
     */
    public static void printAccountState(){
        for(String key: getStateMap().keySet()){
            System.out.println("Adress: " + key + ", " + "Value:
                 " + stateMap.get(key).toString());
        }

    }

    /**
     * Add an AssetType.
     * @param assetType - The assetType to add.
     */
    public static void addAssetType(AssetType assetType) throws
        AlreadyBoundException {
        for(AssetType existingType : assetTypes){
            if(existingType.getID().equals(assetType.getID())){
                return;
            }
        }
        assetTypes.add(assetType);
    }


    /**
     *  Add transaction to pending transactions
     *  @param transaction
     */
    public static void addPendingTransaction(Transaction
        transaction){
        pendingTransactions.add(transaction);
    }

    /**
     * Get the AssetType list
     */
    public static ArrayList<AssetType> getAssetTypes(){
        return assetTypes;
    }

    /**
     * Get the stateMap
     * @return stateMap
     */
    public static HashMap<String, AccountState> getStateMap() {
        return stateMap;
    }

    /**
     *  Set the stateMap
     *  @param stateMap
     */
    public static void setStateMap(HashMap<String, AccountState>
         stateMap) {
        StaticRepository.stateMap = stateMap;
    }

    /**
     *  Add an account with an empty accountState to state
     *  @param address - the address of the account to add.
     **/
    public static void addAccount(String address){
        if(!stateMap.containsKey(address))
            stateMap.put(address, new AccountState());

    }

}
```

A.1.13 TransactionValidator

```
package core;

import exceptions.TransactionNotValidException;

/**
 * This class run in a separate thread to periodically execute
 * and broadcast transactions.
 * @author Ludvig Backlund
 *
 */

public class TransactionValidator {

    public static void main(){

        System.out.println("TransactionValidator initialized");

        while(true){
            try
            {
                StaticRepository.executePendingTransactions();
            }
            catch(TransactionNotValidException e){
                System.err.println("\r" + e.getMessage());
            }

            try {
                Thread.sleep(3000); // Set here how often to
                    execute pending transactions.
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }

}
```

A.1.14 UserInterface

```
package core;

import java.io.*;
import java.rmi.AlreadyBoundException;

import org.apache.commons.codec.DecoderException;
import org.apache.commons.codec.binary.Hex;

/**
 * Command line interface for a user to interact with the
     blockchain.
 *
 * Commands:
 * create wallet - creates a new wallet file containing your
     public and private key.
 * create security - create a new security. Shares are assigned
     to the active wallet.
 * create transaction - create and broadcast a new transaction.
 * print state - prints the current state of the blockchain
 * set wallet - In case several wallet files exists this
     commando allows for deciding which to be active.
 * my address - prints the address of the active wallet.
 * my balance - prints the balance of the active wallet
 *
 * @author Ludvig Backlund
 */


/**
 * Construct a new UserInterface.
 *
 */
public class UserInterface {

    /**
     * The wallet to use when creating Transactions.
     */
    private static Wallet myWallet = null;

    /**
     * Initialize the Command Line Interface.
     */
    public static void initialize(){
        String input = "";
        System.out.println("Enter command:");
        BufferedReader bufferedReader = new BufferedReader(new
            InputStreamReader(System.in));
        try {
            input = bufferedReader.readLine();
        } catch (IOException e) {
            System.err.println("Something went wrong");
            initialize();
        }

        input = input.toLowerCase();

        /** Create wallet */
        if(input.equals("create wallet")){
            createWallet(bufferedReader);
        }

        /** Create transaction */
        if(input.equals("create transaction")){
            createTransaction(bufferedReader);
        }

        /** Emit a security */
        if(input.equals("create security")){
            emitSecurity(bufferedReader);
        }

        /** Set active wallet */
        if(input.equals("set wallet")){
            setWallet(bufferedReader);
        }

        /** Print the current state of all known accounts */
        if(input.equals("print state")){
            StaticRepository.printAccountState();
        }

        /** Print the address of the wallet currently active */
        if(input.equals("my address")){
            if(myWallet != null)
                System.out.println(myWallet.getStringAddress());
            else
                System.err.println("No active wallet set");
        }

        /** Print the balance of the current wallet */
        if(input.equals("my balance")){
            if(myWallet != null)
                System.out.println(StaticRepository.getStateMap
                    ().get(myWallet.getStringAddress()));
            else
                System.err.println("No active wallet set");
        }

        /**
```

```java
         * Print details about an AssetType
         */
        if(input.equals("print assettype")){
            printAssetType(bufferedReader);
        }

        if(input.equals("get nonce")){
            System.out.println(StaticRepository.getStateMap().
                get(myWallet.getStringAddress()).getNonce());
        }

        UserInterface.initialize();
    }

    /**
     * Create a new wallet.
     * @param bufferedReader
     */
    private static void createWallet(BufferedReader
        bufferedReader){

        System.out.println("Enter wallet name");
        String walletName = "";
        try{
            walletName = bufferedReader.readLine();
        } catch(IOException e){
            System.err.println("Something went wrong");
            initialize();
        }
        Wallet wallet = new Wallet(walletName);
        StaticRepository.addAccount(wallet.getStringAddress());
        myWallet = wallet;
        EmptyTransaction transaction = new EmptyTransaction(
            myWallet.getAddress());
        transaction = transaction.signTransaction(myWallet.
            getKey());
        /** Add transaction to repository */
        StaticRepository.addPendingTransaction(transaction);
    }

    /**
     * Create a new security
     * @param bufferedReader
     */
    private static void emitSecurity(BufferedReader
        bufferedReader){

        String sender = "";
        String type = "";
        String name = "";
        String ID = "";
        int quantity = 0;
        Wallet senderWallet = myWallet;

        if(myWallet == null){
            System.out.println("Type in name of wallet to stand
                as emitter.");
            try {
                sender = bufferedReader.readLine();
            } catch (IOException e) {
                System.err.println("Something went wrong");
                initialize();
            }
            File walletFile = new File("sender");
            if(!walletFile.exists()){
                System.err.println("Wallet does not exist.");
                initialize();
            }
            myWallet = Wallet.load(sender);
        }

        System.out.println("Type in name of security to create")
            ;
        try {
            name = bufferedReader.readLine();
        }catch (IOException e) {
            System.err.println("Something went wrong");
            initialize();
        }

        System.out.println("Type in ID of security to create");
        try {
            ID = bufferedReader.readLine();
        }catch (IOException e) {
            System.err.println("Something went wrong");
            initialize();
        }

        System.out.println("Type in type of security to create (
            i.e. 'bond', 'stock', etc.)");
        try {
            type = bufferedReader.readLine();
        }catch (IOException e) {
            System.err.println("Something went wrong");
            initialize();
        }

        System.out.println("Type in number of shares to emit");
        try {
            quantity = Integer.parseInt(bufferedReader.readLine
                ());
        } catch (IOException e) {
            System.err.println("Something went wrong");
            initialize();
        }

        AssetType assetType = new AssetType(name, ID, type,
            senderWallet.getName(), senderWallet.getKey().
            getPubKey());

        try {
            StaticRepository.addAssetType(assetType);
            } catch(AlreadyBoundException e){
            System.err.println(e.getMessage());
```

```java
            initialize();
        }

        EmittingTransaction transaction = new
            EmittingTransaction(senderWallet.getAddress(),
            assetType, quantity);
        transaction = transaction.signTransaction(senderWallet.
            getKey());
        /** Add transaction to repository */
        StaticRepository.addPendingTransaction(transaction);
        System.out.println("Emitting " + quantity + " " + name +
             " to address: " + Hex.encodeHexString(senderWallet.
            getAddress()));
    }

    /** Lets the user create a transaction by entering the
        receiver
     *  the type and amount. Sender is the active wallet.
     *
     * @param bufferedReader
     */
private static void createTransaction(BufferedReader
    bufferedReader){

        String receiver = "";
        String sender = "";
        String assetID = "";
        int quantity = 0;
        Wallet senderWallet = myWallet;

        /** Print all the known addresses to make it easier to
            send */
        System.out.println("Known addresses: ");
        StaticRepository.printAccountState();

        /** Get info about the transaction from user */
        System.out.println("Type in receiveraddress:");
        try {
            receiver = bufferedReader.readLine();
        } catch (IOException e) {
            System.err.println("Something went wrong");
            initialize();
        }

        if(myWallet == null){
            System.out.println("Type in name of wallet to send
                from:");
            try {
                sender = bufferedReader.readLine();
            } catch (IOException e) {
                System.err.println("Something went wrong");
                initialize();
            }
            File walletFile = new File("sender");
            if(!walletFile.exists()){
                System.err.println("Wallet does not exist.");
                initialize();
            }

            myWallet = Wallet.load(sender);
        }

        System.out.println("Enter assetID");
        try
        {
            assetID = bufferedReader.readLine();
        }
        catch (IOException e)
        {
            System.err.println("Something went wrong");
            initialize();
        }

        System.out.println("Type in quantity to transact:");
        try {
            quantity = Integer.parseInt(bufferedReader.readLine());
            if(StaticRepository.getStateMap().get(myWallet.getStringAddress()).getBalance(assetID) < quantity){
                System.err.println("Not enough balance on account");
                initialize();
            }
        } catch (Exception e) {
            System.err.println("Something went wrong");
            initialize();
        }

        /** Create the transaction (Addresses should be as byte []) */
        byte[] receiverByte;
        try {
            receiverByte = Hex.decodeHex(receiver.toCharArray());
            OrdinaryTransaction transaction = new OrdinaryTransaction(senderWallet.getAddress(), receiverByte, assetID, quantity);
            transaction = transaction.signTransaction(senderWallet.getKey());
            /** Add transaction to repository */
            StaticRepository.addPendingTransaction(transaction);
            System.out.println("Transacting " + quantity + " " + assetID + " to address: " + receiver + ".");
        } catch (DecoderException e) {
            System.err.println("Something went wrong, can't decode receiver address.");
            initialize();
        }
    }

    /**
     * Set the active wallet
     * @param bufferedReader
     */
    private static void setWallet(BufferedReader bufferedReader)
        {
        String walletName = "";
        System.out.println("Type in name of your wallet");
        try {
            /** Read wallet name from user */
            walletName = bufferedReader.readLine();

            /** Check that wallet exists */
            File walletFile = new File(Blockchain.getDirectory()
                 + walletName);
            if(walletFile.exists()){
                myWallet = Wallet.load(walletName);
                StaticRepository.addAccount(myWallet.
                    getStringAddress());
                System.out.println("Active wallet set to: " +
                    walletName);
            } else {
                System.err.println("Could not find wallet: " +
                    walletName);
            }
        } catch (IOException e) {
            System.err.println("Something went wrong");
            initialize();
        }
    }

    /**
     * Print information about an AssetType.
     * @param bufferedReader
     */
    private static void printAssetType(BufferedReader
        bufferedReader) {

        String assetName = "";
        System.out.println("Type in name of AssetType:");
        try{
            assetName = bufferedReader.readLine();
            for(AssetType assetType: StaticRepository.
                getAssetTypes()){
                if(assetType.getName().equals(assetName)){
                    System.out.println(assetType.toString());
                    return;
                }
            }
            System.err.println("Could not find asset: " +
                assetName );
        } catch(IOException e){
            System.err.println("Something went wrong");
            initialize();
        }

    }
}
```

A.1.15 Wallet

```java
package core;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.io.UnsupportedEncodingException;
import java.math.BigInteger;

import org.apache.commons.codec.binary.Hex;

import crypto.ECKey;

/**
 * Class representing a wallet. Contains
 * a users private and public key.
 * @author Ludvig Backlund
 *
 */

public class Wallet implements Serializable {

    private static final long serialVersionUID = 1L;

    private BigInteger priv;
    private transient ECKey key;
    private String walletName;
    private byte[] address;

    public Wallet(String walletName){
        this.walletName = walletName;
        this.key = new ECKey();
        this.priv = key.getPrivKey();
        this.address = key.getAddress();
        Wallet.store(this);
    }

    public Wallet(String walletName, ECKey key) throws
        UnsupportedEncodingException{
        this.walletName = walletName;
        this.priv = key.getPrivKey();
        this.key = key;
        this.address = key.getAddress();
    }


    /** Store a wallet on disk.
     *
     */
    public static void store(Wallet wallet){
        try {
            File file = new File(Blockchain.getDirectory() +
                wallet.getName());
            FileOutputStream fileOut = new FileOutputStream(file
                );
            ObjectOutputStream out = new ObjectOutputStream(
                fileOut);
            out.writeObject(wallet.getPriv());
            out.close();
            fileOut.close();
            System.out.println("Wallet saved as: " + wallet.
                getName());
        } catch(IOException i)
          {
              i.printStackTrace();
          }
    }

    /** Load wallet from disk, if wallet can't be found a new
        wallet is created.
     *
     * @param walletName - name of wallet to load.
     * @return loadWallet if found else null.
     */
    public static Wallet load(String walletName){
        File walletFile = new File(Blockchain.getDirectory() +
            walletName);
        System.out.println("Searching for wallet " + walletFile.
            getAbsolutePath());
        if(walletFile.exists()){
            try{
                FileInputStream fileIn = new FileInputStream(
                    walletFile);
                ObjectInputStream in = new ObjectInputStream(
                    fileIn);
                BigInteger loadPriv = (BigInteger) in.readObject
                    ();
                in.close();
                fileIn.close();
                // (Some compression stuff to make addresses
                    correct)
                ECKey key = ECKey.fromPrivate(loadPriv);
                Wallet loadWallet = new Wallet(walletName, key);
                return loadWallet;

            }catch(Exception i)
            {
                UserInterface.initialize();
                //i.printStackTrace();
            }
            return null;
        }
        else{
            System.out.println("Couldn't find wallet, new wallet
                 created as: " + walletName);
            return new Wallet(walletName);
        }
    }

    public String getName(){
        return walletName;
```

```java
    }

    public BigInteger getPriv(){
        return priv;
    }

    public ECKey getKey(){
        return key;
    }

    public byte[] getAddress(){
        return address;
    }

    public String getStringAddress(){
        return Hex.encodeHexString(address);
    }

}
```

A.2 Package: network

A.2.1 MultipleSocketServer

```java
package network;

import java.net.*;
import java.util.ArrayList;
import java.util.List;
import java.io.*;
import core.*;

/**
 * Server class that can handle multiple sockets.
 * Used for receiving data from the network.
 *
 * @author Ludvig Backlund
 */

public class MultipleSocketServer implements Runnable {

    private Socket connection;
    private int id;
    static String ip;

    public static void main(){

        int port = 19999;
        int count = 0;

        try
        {
            InetAddress addr = InetAddress.getByName(ip);
            ServerSocket socket1 = new ServerSocket(port, 50,
                addr);

            System.out.println("MultipleSocketServer Initialized
                 - ip: " + ip);

            while(true){
                Socket connection = socket1.accept();
                Runnable runnable = new MultipleSocketServer(
                    connection, ++count);
                Thread thread = new Thread(runnable);
                thread.start();
            }
        }
        catch(Exception e)
        {
            System.err.println("Couldn't start server.");
        }
    }

    MultipleSocketServer(Socket socket, int id){
        this.connection = socket;
        this.id = id;
    }

    public static void setIP(String ip){
        MultipleSocketServer.ip = ip;
    }

    public static String getIP(){
        return ip;
    }

    public void run(){

        List<String> receiverAddress = new ArrayList<>();
        Object[] objectsToSend = new Object[2];
        try
        {
          BufferedInputStream is = new BufferedInputStream(
              connection.getInputStream());
          ObjectInputStream ois = new ObjectInputStream(is);
          Object object = ois.readObject();

          /** Handle receiving a Transaction */
          if( object instanceof Transaction){
              Transaction receivedTransaction;
              if(object instanceof EmptyTransaction){
                 receivedTransaction = (EmptyTransaction) object
                     ;
              }
              else if(object instanceof EmittingTransaction){
                 receivedTransaction = (EmittingTransaction)
                     object;
              }
              else{
                 receivedTransaction = (OrdinaryTransaction)
                     object;
              }
              System.out.println("Transaction of " +
            receivedTransaction.getAmount() + " " +
              receivedTransaction.getAssetID() + " from:
                 " + receivedTransaction.
                getSenderAddress() +
          " to " + receivedTransaction.getReceiverAddress()
            + " received.");
          StaticRepository.addPendingTransaction(
            receivedTransaction);
        }

        /** Handle receiving a Block */
        if(object instanceof Block){

          Block receivedBlock = (Block) object;

          if(receivedBlock.validateBlock()){
            ArrayList<Transaction> transactions =
              StaticRepository.getUncommonTransactions(
              receivedBlock);
            Blockchain.addBlock(receivedBlock);
            new StaticRepository(transactions);
          }
        }

          /** Code for blockchain synchronization*/
        if(object instanceof String[]){
          String[] id = (String[]) object;

         /** Peer is telling us blockchain is synced (
           returnCode[0] = 1) */
         if(id[0].equals("1")){
           Blockchain.setSynchronized(id[1], true);
           System.out.println("Synced with " + id[1] + " "
             + Blockchain.getIsSynchronized(id[1]));
         }

         /** Peer is requesting a block */
         if(id[0].equals("0")){
           Block block = Blockchain.getBlockByPrevHash(id
             [1]);
           receiverAddress.add(id[2]);
           objectsToSend[0] = receiverAddress;
           /** Check if block was found */
           if(block != null){
             objectsToSend[1] = block;
           }
           /** If no block was found tell peer using a
             return code (returnCode[0] = 1)*/
           else{
             String[] returnCode = new String[2];
             returnCode[0] = "1";
             returnCode[1] = ip;
             objectsToSend[1] = returnCode;
           }
           /** Send answer to peer. */
           Client.main(objectsToSend);
         }
```

```
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        finally {
            try
            {
                connection.close();
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

A.2.2 Client

```
package network;
import java.net.*;
import java.util.List;
import java.io.*;
import core.*;
import util.InfoFile;

/**
 * Client class called on when transmitting
 * objects over the network. Three types of objectcs
 * can be transmitted; Transactions, Blocks and String arrays.
 * String array is only used for blockchain sync between peers.
 * @author Ludvig Backlund
 *
 */

public class Client {


     public static void main(Object[] args) {
            /** Define a port */
            int port = 19999;

            /** Get the addresses to transmit to */
            List<String> addresses = (List<String>) args[0];
            if(addresses.contains("broadcast"))
                addresses = Blockchain.getLatestBlock().
                    getValidPeers();

            /** Get the object to transmit */
            Object obj = args[1];

            /** Check what kind of object we are transmitting */
            if(obj instanceof Block){
                obj = (Block) obj;
            }
```

```java
           if(obj instanceof Transaction){
               /** Don't re-send to nodes.*/
               Transaction trans = (Transaction) obj;
               addresses.removeAll(trans.getSenderIP());
           }

           /** Broadcast to peers */
           for(String peer: addresses){

               /** Don't send to self */
               if(peer.equals(MultipleSocketServer.getIP()))
                   continue;
              try
              {
                   /** Obtain an address object of the server */
                   InetAddress address = InetAddress.getByName(peer);

                    /** Establish a socket connection */
                   Socket connection = new Socket(address, port);

                   /** Instantiate OutputStream objects */
                   BufferedOutputStream bos = new BufferedOutputStream(connection.getOutputStream());
                   ObjectOutputStream oos = new ObjectOutputStream(bos);

                   /** Send object */
                   oos.writeObject(obj);
                   oos.flush();

                   /** Close the socket connection. */
                   connection.close();

              } catch(NullPointerException e)
              {
                  e.printStackTrace();
              }
               catch(IOException e)
              {
                   /** Set peer as synchronized since we can't connect to it */
                   Blockchain.setSynchronized(peer, true);
              }
               catch(Exception e)
              {
                   System.out.println("Exception: " + e.toString());
                   e.printStackTrace();
              }
           }
       }
    }
```

A.2.3 Synchronize

```java
package network;

import java.util.ArrayList;
import java.util.List;

import core.*;
import util.InfoFile;

/**
 * Class called on start-up for synchronizing our blockchain
     with peers.
 * Synchronization is done by requesting a block that has our
     latest
 * block as its previous block hash. If peer can't find such a
     block it
 * the peer is set as synchronized.
 * @author Ludvig Backlund
 *
 */
public class Synchronize {

    /** Height of peers blockchain */
    static int peerHeight;


    public static void main(){

        /** Set all addresses to unsynced */
        for(String address : Blockchain.getLatestBlock().
            getValidPeers())
            Blockchain.setSynchronized(address, false);

        /** Synchronize blockchain */
        synchronize();

        System.out.println("Blockchain synchronized: Our height
            = " + Blockchain.getHeight());
    }


    /** id[0] = 0, Request next block.
     *  id[1] = block-hash, Hash of our latest block.
     *  id[2] = Our server-ip,
     **/
    public static void synchronize(){
        boolean synched = false;
        while(!synched){
            /** Create a String[] with necessary info */
            Block block = Blockchain.getLatestBlock();
            String[] id = new String[3];
            id[0] = "0";
            id[1] = block.getHash();
            id[2] = MultipleSocketServer.getIP();

            /** Broadcast to network */
            List<String> receivers = new ArrayList<>();
            receivers.add("broadcast");
            Object[] objectsToSend = {receivers, id};
            Client.main(objectsToSend);

            /** Wait litte bit...*/
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            /** Check if synchronized */
            synched = true;
            for(String peer : Blockchain.getLatestBlock().
                getValidPeers()){
                if(!Blockchain.getIsSynchronized(peer) && !peer.
                    equals(MultipleSocketServer.getIP())){
                    synched = false;
                }
            }
        }
    }
}
```

A.3 Package: start

A.3.1 Start

```java
package start;

/**
 * @author Ludvig Backlund
 *
 * Commands:
 * create wallet - creates a new wallet file containing your
     public and private key.
 * create security - create a new security. Shares are assigned
     to the active wallet.
 * create transaction - create and broadcast a new transaction.
 * print state - prints the current state of the blockchain
 * set wallet - In case several wallet files exists this
     commando allows for deciding which to be active.
 * my address - prints the address of the active wallet.
 * my balance - prints the balance of the active wallet
 */

public class Start {
    /** ONLY CHANGE USER WHEN STARTING, CURRENTLY USER 0-3 IS
        ACCEPTED */
    private static int user = 1;

    private static String directory;
    private static String ip;

    public static void main(String[] args){


        /******************************************************
            */

        /** If absolute path is wanted */
        String directoryAbs = "";

        //directories = new String[nrUsers];
        //ip = new String[nrUsers];
        //isGenesis = new String[nrUsers];

        /** Set the directory for each user as directory/user0,
            directory/user1 and so on... */
        directory = directoryAbs + "user" + user + "/";

        /** Set the ip of each user as 127.0.0.1, 127.0.0.2 and
            so on... */
        ip = "127.0.0." + (user+1);


        /** Start a user in a separate thread using the start
            arguments */
        new Thread(new Start().new UserHandler()).start();
        try {
            Thread.sleep(15000);
        } catch (InterruptedException e) {

            e.printStackTrace();
        }



    }
        private class UserHandler implements Runnable{


            public UserHandler(){
            //System.out.println("From constructor: " + Thread.
                currentThread().getName() + " " + directories[
                userNr] + " " + ip[userNr]);
            }
            public void run(){
                System.out.println("User " + user + " joining.")
                    ;
                String[] input = {directory, ip};
                User.main(input);
            }

        }

}
```

A.3.2 User

```
package start;
import core.*;
import network.*;
```

```java


public class User{

    public static void main(String[] args){

        new Blockchain();
        MultipleSocketServer.setIP(args[1]);
        Blockchain.setDirectory(args[0]);

        ThreadGroup threadGroup = new ThreadGroup(args[1]);

        /** Load the blockchain */
        Blockchain.load();

        /** Create repository */
        new StaticRepository(Blockchain.getLatestBlock());

        /** Start the server in a separate thread */
        Thread server = new Thread(threadGroup, new
            ServerHandler());
        server.start();


        /** Synchronize the blockchain (not genesis creator) */
            Thread sync = new Thread(threadGroup, new
                SyncHandler());
            sync.start();
            System.out.println("Synchronizing blockchain...");
            try {
                sync.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }


        /** Load the blockchain */
        Blockchain.load();

        /** Create repository */
        new StaticRepository(Blockchain.getLatestBlock());


        /** Start the miner in another tread */
        Thread miner = new Thread(threadGroup, new MinerHandler
            ());
        miner.start();

        /** Start transaction validator */
        Thread transactionValidator = new Thread(threadGroup,
            new TransValHandler());
        transactionValidator.start();

        /** Fire up the CLI */
        Thread CLI = new Thread(threadGroup, new CLIHandler());
        CLI.start();
    }

    private static class ServerHandler implements Runnable{

        public void run(){
            MultipleSocketServer.main();
        }

    }

    private static class MinerHandler implements Runnable{

            public void run(){
                Miner.main();
            }
        }

    private static class CLIHandler implements Runnable{

        public void run(){
            UserInterface.initialize();
        }
    }


    private static class TransValHandler implements Runnable{

        public void run(){
            TransactionValidator.main();
        }
    }
    private static class SyncHandler implements Runnable{

            public void run(){
                Synchronize.main();
            }
        }
}
```