Blockchain Systems General Design And Scorex

Alexander Chepurnoy

June-August, 2016

Chapter 1

Introduction

This is the guide to the blockchain data structure and digital currencies built on top of it. Unlike previous guides we do not start with an example of Alice sending a signed note to Bob. Instead, we describe data structures and a process of building them in a decentralized environment. The guide is made for people understanding basic data structures like sets and append logs. No cryptography knowledge is needed.

We assume a reader already has general knowledge of Bitcoin. We also assume a reader has a general knowledge in most known topics of Cryptography, such as public key cryptography and cryptographic hash functions. For good explanation of cryptography primitives and protocols please refer to the foundational book of [**?**].

In the first chapter we describe basic building blocks of a blockchain system. In the second chapter we describe how the blocks are implemented in the Scorex framework. We provide code snippets in Scala language. No prior knowledge of the Scala language is required.

1.1 Background

1.2 Other Tutorials

"How the Bitcoin protocol actually works" tutorial by Michael Nielsen [**?**] define problems led to the creation of Bitcoin and how the cryptocurrency solved them. The original Bitcoin whitepaper by Satoshi Nakamoto [**?**] still provides very good description of basic concepts of Bitcoin. Details of Bitcoin technicalities could be found in the "Mastering Bitcoin" book [**?**].

Chapter 2

A Blockchain System Design

2.1 Introduction

Blockchain is a prefix-immutable append log of non-conflicting authenticated events happen in a decentralized peer-to-peer network. Simple, eh? *But what all this actually means?*

Simply said, there are peers do not trust each other. There is no any trusted party, only a protocol peers need to follow (being effectively thrown away from the network otherwise). Peers are issuing authenticated (signed) events of some semantics. For example, they are sending out signed payments. Or they are registering $name \rightarrow value$ correspondences in a shared database (certificates, domains). "Prefix-immutable append log" means all the peers following the protocol are agree on immutable prefix of events append log. That is, if we cut a suffix of some length from the log a peer holds, for each peer, same-size prefixes will be the same, with overwhelming probability. Events in the ordered prefix-immutable log must be non-conflicting in order to have flawless history.

Consider Bitcoin as an example. Peers are holding money in form of algorithmically issued tokens. They do not trust each other, and do not to seek for a trusted mediator. Instead, they are running a Bitcoin protocol which builds prefix-immutable append log containing token transfers. Last few versions of the log are considered potentially unstable, but before them the history is considered as irreversible. The payments history is flawless, so, for example, if Alice had sent all her tokens to Bob, she can't send anything after that and before receiving tokens from other party.

2.2 Cryptography

[TODO: public key cryptography]

[TODO: hash functions]

2.3   Transactional Layer

In this section we define a generalized view of transactional semantics of a blockchain system. The two foundational concepts here is a *state* and a *transaction*.

2.3.1   Minimal State

Consider a transaction arrived at a node. The node is doing following on receiving it:

1. Checks whether a transaction is valid

2. Apply it if so

Intuitively, there are some stateless checks, e.g. whether a signature for a transaction is valid, whether amount of tokens to transfer is non-negative, but also there are stateful ones. For example, if Alice is sending tokens to Bob, a node must be sure Alice has enough funds in order to make a payment. Or, if Alice is registering a domain, a node must be sure it is not taken yet.

So a node needs to store some state in order to validate incoming transactions. And there is some *minimal state* representation enough to validate an arbitrary transaction while removing any element from the representation eliminating this property. So all the nodes share this minimal state but a node could also store some additional information.

By applying a transaction a minimal state is being modified. It should be impossible to apply a transaction already processed.

For many reasons almost all cryptocurrencies of today are packing transactions into *blocks*. We can think about a block as of *atomic batch state update*.

[TODO: Block header - tx part]

We can state some axioms here.

Axiom 1. *There is some initial state hard-coded into each node. Further we name it genesis state.*

Axiom 2. *Validation and application of a transactions(and possibly an additional metadata) are deterministic procedures. All the honest nodes follow the same rules.*

Proposition 1. *If the same sequence of blocks is applied to the genesis state for two different nodes, then the resulting minimal states will be the same.*

Proof. Consider the nodes have the same minimal state and trying to apply the same block to it. By the Axiom 2, they will have the same minimal state as result, as verification and application procedures are deterministic. By the

Axiom 1, genesis state is the same for all the nodes. By induction, result of sequential applying of the blocks results in the same minimal states for all the nodes. □

Further we will use both the terms "minimal state" and "state" interchangeably.

2.3.2 Bitcoin

In Bitcoin a transaction contains multiple *inputs* and *outputs*. Inputs are connected with outputs of transactions previously applied to a state, and the connected outputs must not be spent yet. That is, the outputs to be connected by the inputs of the transactions do not have connections from transactions previously applied to the state. Thus an output could be spent as whole only and so we can consider a set of unspent outputs as a minimal state.

How to spend an output? In Bitcoin it contains a script in a stack-based language. Input also contains a script. Then an input could spend an output if a combined script made of inputs' and then outputs' could be executed and results in non-zero top stack item.

[TODO: example]

2.3.3 Boxes, Propositions and Proofs

Abstracting the Bitcoin-like model, a minimal state could be represented as a set of *closed boxes* of size n_S. Each box has a value associated with it. Say, a transaction opens n_k boxes and also creates n_b new closed boxes, then the resulting state set has the size of $n_S - n_k + n_b$ after applying the transaction to it.

How to open a box? We can protect a box with a script in Bitcoin language. Or we can put a public key into closed box and then it is possible to open it with a proof of private key knowledge, a signature (we will consider details further). To describe these approaches as well as many others possible in a general way, we say a box is protected by a *proposition* of some kind, and in order to open it, a *proof* of the same kind must be provided. There are some tricky details we will discuss further.

2.3.4 Namecoin

Namecoin, in addition to token transfers, introduce *name* → *value* storage. In general, values could be arbitrary, but there are few standard namespaces with predefined semantics, for domains and identities.

We do not specify Namecoin design precisely below, but some Namecoin-like design.

Consider a transaction contains a box with *name_register* command specifying a *name* → *value* correspondence. Such a box has zero value and associated with a public key *pk*. It is demanded to pay some fee in order to put such a

box into state. The box lives in the state for some period of time, then it is considered as expired and could be thrown away from a minimal state. It is possible to renew or transfer ownership to a different public key by publishing a *name_update* box replacing an original one in the state.

This design has a critical flow. A block generator could refuse to include *name_register* command into a block and put its own value for the same name. This is an example of *frontrunning attack*, when an original transaction is suppressed by another one issued by an attacker. In order to avoid frontrunning attacks, Namecoin has *name_new* command to announce the intention to register a name by providing its hash value.

2.3.5   Nxt and Ethereum

Having a Bitcoin wallet, you can be goggled by complexity of boxes and propositions in form of stack-based scripts. Why not to have just accounts and token transfers between them instead?

Actually some of Bitcoin successors walked this path. For example, Nxt has a dedicated notion of accounts. An account is associated with its public key. A transaction transfers tokens from one account to another and needed to be signed by the sender. For such a system stateful verification needs for a minimal state in form of table holding a correspondence between accounts and their balances.

With such a simple minimal state design we have a problem though. Let's describe it with an example. Alice has 50 tokens at some moment of time. She issues a signed transaction to pay 5 tokens to Bob. A node can validate the transaction and found it valid, and so applicable. After the application Alice has 45 tokens. But how to prevent second application of the transaction? Our minimal state representation seems to be flawed.

Ethereum solves the problem by modifying minimal state representation adding "nonce" value to it. That is, minimal state is not about (public key $\rightarrow$ balance) correspondence anymore, but (public key $\rightarrow$ (nonce, balance)). Transaction contains nonce value *txnonce* as well, and transaction is valid and so applicable only if $txnonce = nonce + 1$. By application, $nonce := txnonce$.

Unlike Bitcoin, Ethereum sets strict order of transactions issued by an account. In Bitcoin, transactions could be applied in any order, if they are spending non-overlapping sets of outputs, and input of one transaction does not spend an output of another. In Ethereum, order of transaction is set by nonce values.

2.3.6   Transactional Metadata

Assume we have a set of objects serializable to a set of unique byte arrays. We want to *authenticate* these binary representations in an efficient. That is, we want to calculate a fixed-sized value for a whole set such as a single bit change always results in a change of the *authenticating* (or *root*) value, and the value is collision-resistant, so it is impossible(with non-negligible probability) to generate different set resulting in the same root value.

[TODO: Merkle tree / authenticated data structures explanation]

Along with transactions, we can put some aggregated data about them. For example. in Bitcoin's block a root hash of a Merkle tree for the transactions in block is put into the block. That way it is possible for nodes in a network to exchange not full blocks but *blockheaders*. A *blockheader* is a block without its transactions. By including transactions root hash into the blockheader it is possible to have it spread around a network and be sure it is impossible to show transactions set other that was included.

[TODO: Bitcoin example]

2.3.7 Transactional Layer Generalization

After the examples, let's summarize what we have in common in all the observable cryptocurrencies.

1. **A Proposition And A Proof.** In an every imaginable blockchain we have objects to be protected by secret owners. To achieve the property of being protected we introduce a proposition, and an object could be modified or destroyed only by presenting a proof satisfying a proposition. There are a lot of possible instantiations, e.g. Bitcoin scripts or digital signatures.

2. **Box Structure.** There is a minimal element of a replicated state we are calling a *box*. A box is protected by a proposition. It is possible to modify it or destroy it only by showing a proof satisfying a proposition.

3. **Minimal State.** Minimal state is a most compact structure giving an ability to verify a transaction against it. Minimal state is about a set of boxes.

4. **Transaction And Transactional Language.** Transaction is a smallest possible atomic state modifier. A transaction is to be verified against a state in a deterministic fashion (so given a state and a transaction, two nodes will always give the same validation result whether *true* or *false*). If a transaction is valid against a state it could modify the state. Validation and application rules are individual (Ethereum even brings quasi Turing completeness here).

5. **Block.** All the blockchain systems are storing transactions in full blocks. Most of them also have some authenticating value for the set of block transactions included into block thus it is possible to use block headers instead of full blocks in many scenarios in order to reduce a load.

[TODO: wallet section?]

2.4 Consensus

We have proven(in the proposition [?]) that if the same sequence of blocks carrying transactions is applied to the same genesis state for two node then they will have the same state. It is exactly what do we want to achieve, but how to have the same sequence of blocks for all the nodes?

In the first place, we can achieve this only for nodes willing to achieve this by following some protocol strictly. We refer to such nodes as to *honest* nodes, and to the protocol as to *consensus protocol*. If nodes are not following the protocol we call them *byzantine* nodes. A byzantine node could be malicious, but also it could be not able to follow the consensus protocol because of software bugs, problems with connectivity, misleading information sent from outer world etc.

[validity, agreement, termination]

Computer Science studies consensus protocols since early 1980s. A lot of interesting results were generated in this field. For example, it is impossible to achieve consensus using a deterministic procedure for a set of nodes if they are exchanging messages asynchronously and a single process could fail(Fischer-Lynch-Paterson theorem []).

Consensus in open networks, so with unknown number of participants, is pretty new and very hard question.

1. Validity

2. Agreement

3. Termination

For a blockchain consensus protocols, we can state following properties:

1. Consistency(or Prefix immutability) - for two honest nodes the probability to have different prefixes after cutting last k blocks should go down with k and be negligible after some value. The good option is to have the probability going down exponentially with k.

2. Chain Quality - a party having $x\%$ of voting power should produce no more than $(\alpha \cdot)x\%$ blocks in a long run, where α is constant.

3. Chain Growth - over time blockchain should always grow. No one is interested in a structure with possibility to stuck.

2.4.1 Proof-of-Work

Proof-of-Work consensus protocol introduced in the foundational paper of Bitcoin [?] is in the core of Bitcoin, Ethereum as well as many other cryptocurrencies. The basic idea of the protocol is to force miners to iterate over output of some function with a small probability of success per iteration. A successful result is giving a right to generate a block. The probability is adjusted automatically via *difficulty* parameter D.

In case of Bitcoin, the function is just a hash function, but what is about its input?

We want to make blocks immutable after creation. For that, we are applying hash function for all the block contents.

[TODO: Merkle tree]

We also want for a block to refer to a previous block. So we include a hash of a previous block, in this case it in order to replace a block with another one it is needed to replace all its descendants also. As some amount of work is needed to generate each block in order to replace a chain suffix of length l, amount of work proportional to l is needed.

With Proof-of-Work it is impossible to make a false claim on successful block generation. Such a claim is easily verifiable, as calling hash function is very cheap. Thus a Proof-of-Work works as a protection against Sybil attacks [].

2.4.2 Proof-of-Stake

There are some disadvantages of Proof-of-Work schemes. A lot of resources to be spent. During early days a Proof-of-Work currency could be destroyed by a miner already having a lot of computational resources working for another blockchain.

Can we have a protection against Sybil attacks without spending computational resources? The simplest way to achieve this is to use cryptocurrency tokens as anti-Sybil tools. That is, a probability to generate a block is proportional to a stake a node holds.

2.4.3 History

Previously we talked about a blockchain. But in all the global networks collisions are possible (to prevent them we need to have a global lock or synchronous rounds and then some kind of leader election). So in a network a blocktree lives.

[TODO: a blocktree picture]

In most cases the fact of blocktree existence in omitted. A node is storing a blockchain. A blockchain has some score(e.g. a chain length, but this is a totally insecure scoring function, see [**?**] for details). If better chain is declared in network, a node is throwing away blocks until common block and then apply better suffix. In this case a node sees a blocktree only during switching from one branch of it to another. There are some proposals to explicitly use a blocktree. For example, in GHOST scoring function [] a chain with heaviest tree wins.

2.4.4 Consensus Layer Generalization

2.5 Peer-to-Peer Network

Blockchain is maintained in a peer-to-peer network. For simplicity we are starting with a network where all the nodes

2.6 Incentives

2.7 Complications

fully prunable outputs ZCash

2.8 Conclusion

2.9 Further Reading

Proof-of-Work and blockchain were introduced in the foundational Bitcoin whitepaper [**?**].

Overview of Bitcoin P2P layer along with description of possible Eclipse attacks(partly fixed to the moment) against it could be found in [**?**]

Chapter 3

Scorex

3.1 Introduction

Scorex is a modular blockchain core framework. It supports definitions given in the previous chapter in form of Scala code. What do you need to do in order to build something on Scorex, is to implement all the abstract interfaces, or just some of them and use ready modules for missing parts.

3.1.1 *Scala Language

We will describe some concepts of Scala language in sections started with "*". If you already a Scala developer, you can miss the sections. Experience with programming languages(say, Java, C++, OCaml or Rust) is needed, as we will explain Scala features used in code snippets provided very quickly. For a good introduction into the Scala language, please refer to [**?**].

Scala is functional, modular and also object-oriented language. Such a mix of concepts means different developers can follow very different styles.

[TODO: quick description]

3.2 Transactional Layer

Transactional layers describes blockchain semantics. That is, what is a minimal state enough to validate incoming transactions, what is transaction and how processing of it affects the state, how transactions are protected from being spent by non-allowed party, how wallet is organized etc

3.2.1 *Scala: Traits and Type Parameters

The basic piece in a sufficiently large Scala codebase is a *trait*. Trait is an abstract(so non-instantiable) interface describing functions and values a concrete implementation(class or object) must provide to its users. We can also

think about trait as of type and also a parametrized module. A trait could be parametrized not only by values, but also by types.

Quick example: [TODO: example]

3.2.2 Propositions and Proofs

In the first place, we are getting into a mechanism to protect binary objects, e.g. transaction outputs, from non-permissioned access. We protect an object with a *proposition*. Then in order to make an action with an object it is needed to provide a *proof* for its proposition.

Proposition is very abstract concept. The only property we require from it is to be serialized into bytes. In form of Scala code it is described as:

```
trait Proposition extends BytesSerializable
```

In most useful scenarios a proposition should be addressable. That is, it could be addressed by some identifier, or identifiers.

A proof is an object which could satisfy a proposition given an additional input, namely a *message*(e.g. transaction bytes). As well as a proposition, a proof could be serialized into bytes.

```
trait Proof[P <: Proposition] extends BytesSerializable {
  def isValid(proposition: P, message: Array[Byte]): Boolean
  ...
}
```

3.2.3 Box

A box is a minimal state element. An unspent output in Bitcoin is a box. An account in certain state in Nxt or Ethereum is also a box. Basically, a box is about some value(how many system tokens are associated with it), and some proposition which protects a box from being spent by anyone but a party(or parties) knowing how to satisfy the proposition.

```
trait Box[P <: Proposition] extends BytesSerializable {
  val proposition: P

  val value: Long

  val id: Array[Byte]
}
```

3.2.4 *Scala: Sum Types and Try

3.2.5 A Transaction and Minimal State

As it was shown in the Chapter 1[TODO: link], a transaction and a minimal state could be defined via each other: a transaction is a state modifier, which

also could be valid or not against a state, and applicable if and only if it is valid.
A minimal state is a data structure which deterministically defines whether an
arbitrary transaction is valid and so applicable to it or not.

```scala
abstract class Transaction[P <: Proposition, TX <: Transaction[P, TX]] extend
  def validate(state: MinimalState[P, TX]): Try[Unit]
  def changes(state: MinimalState[P, TX]): Try[StateChanges[P]]


  ...
}
```

We are defining functional interface for minimal state below:

```scala
trait MinimalState[P <: Proposition, TX <: Transaction[P, TX]] {
  def version: Int

  def isValid(tx: TX): Boolean = tx.validate(this).isSuccess

  def processBlock(block: Block[P, _, _]): Try[Unit]
  def rollbackTo(height: Int): Try[Unit]

  def closedBox(boxId: Array[Byte]): Option[Box[P]]


  ...
}
```

So a minimal state knows its current version, can apply a block, and also
rollback to a previous version. To validate a transaction, it just calls *validate*
function of a transaction and checks its result.

3.2.6 Wallet

[TODO:]

3.2.7 Memory Pool

```scala
trait UnconfirmedTransactionsDatabase[TX <: Transaction[_, TX], TData <: Tran
  def putIfNew(tx: TX): Boolean

  def all(): Seq[TX]

  def getById(id: Array[Byte]): Option[TX]

  def packUnconfirmed(): TData

  def clearFromUnconfirmed(data: TData): Unit

  def onNewOffchainTransaction(transaction: TX): Unit
```

```
    def remove ( tx : TX)

}
```

3.2.8   Transactional Block Data

```
trait TransactionalData [TX <: Transaction [_, TX]] extends BytesSerializable w
    val mbTransactions :  Option [ Traversable [TX]]

    val headerOnly = mbTransactions . isDefined
    . . . .
}
```

3.2.9   Transactional Module

3.3   Consensus Layer

3.3.1   Block

```
class Block [P <: Proposition , TData <: TransactionalData [_ <: Transaction [P,
```

3.3.2   History and Blockchain

3.4   Network Layer

Network layer in Scorex is simpler than in Bitcoin or Nxt.

3.4.1   Peer Discovery

3.4.2   Broadcasting Strategies

3.4.3   View Synchronizing

3.5   Ready Modules

There are few modules already implemented.

3.5.1   Proof-of-Stake

Proof-of-Stake module contains implementations of two Proof-of-Stake consensus algorithms.

3.5.2 Simple Transactions Module

Simplest Transactions Module contains an implementation of transactional module with only one kind of transactions, just tokens transfers from one public key to another.

3.5.3 Permacoin Implementation Module

The module contains an implementation of Permacoin consensus protocol [**?**].

3.6 Conclusion