The ΣRGO Yellowpaper

Alexander Chepurnoy   Nazeem Faour   Dmitry Meshkov

May 21, 2018

Contents

Introduction

Multiple Modes

Ergo (since the very first testing network Testnet0) is supporting multiple security models. In addition to full node mode, which is similar to Bitcoin fullnode, Ergo reference implementation supports Light-SPV, Light-Fullnode and Pruned-Fullnode modes.

Full-Node Mode

Like in Bitcoin, a full node is storing all the full blocks since genesis block. Full node checks proofs of work, linking structure correctness (parent block id, interlink elements), and all the transactions in all the blocks. A fullnode is storing all the full blocks forever. It is also holding full UTXO set to be able to validate an arbitrary transaction. The only optimization a fullnode is doing is that is is skipping downloading and checking AD-transformation block part (see below in the "Light-Fullnode" section). For the full node regime, modifiers precessing workflow is as follows:

1. Send ErgoSyncInfo message to connected peers.

2. Get response with INV message, containing ids of blocks, better than our best block.

3. Request headers for all ids from 2.

4. On receiving header:

```
if(history.apply(header).isSuccess) {
    if(!isInitialBootstrapping) Broadcast INV for this header
    Request transaction ids from this block
} else {
    blacklist peer
}
```

5. On receiving transaction ids from header:

```
transactionIdsForHeader.filter(txId => !MemPool.contains(txId)).foreach { txId =>
    request transaction with txId
}
```

6. On receiving a transaction:

```
if(Mempool.apply(transaction).isSuccess) {
    if(!isInitialBootstrapping) Broadcast INV for this transaction
    Mempool.getHeadersWithAllTransactions { BlockTransactions =>
        GOTO 7
    }
}
```

7. Now we have BlockTransactions: all transactions corresponding to some Header

```
if(History.apply(BlockTransactions) == Success(ProgressInfo)) {
    if(!isInitialBootstrapping) Broadcast INV for BlockTransactions
    /*We should notify our neighbours, that now we have all the transactions
    State apply modifiers (may be empty for block in a fork chain)
    and generate ADProofs for them.
    TODO requires different interface from scorex-core,
    because it should return ADProofs
    TODO when mininal state apply Progress info,
    it may also create UTXOSnapshot
    (e.g. every 30000 blocks like in Ethereum).
    This UTXOSnapshot should be required for mining by Rollerchain*/
    if(State().apply(ProgressInfo) == Success((newState, ADProofs))) {
        if("mode"="full" || "mode"=="pruned-full") ADProofs.foreach ( ADProof => History.apply(ADPro
        if("mode"=="pruned-full" || "mode"=="light-full") drop BlockTransactions and ADProofs older
    } else {
        //Drop Header from history, because it's transaction sequence is not valid
```

2

```
        History.drop(BlockTransactions.headerId)
    }
} else {
    blacklist peer who sent header
}
```

Pruned Full-Node Mode

This mode is similar to fast-sync in Geth or Grothendieck, warp-mode in Parity (all the three are Ethereum protocol clients), but makes more aggressive optimizations. In particular, a pruned-fullnode is not downloading and storing full blocks not residing in a target blockchain suffix, and also removing full blocks going out of the suffix. In detail, a pruned client is downloading all the headers, then, by using them, it checks proofs-of-work and linking structure(or parent id only?). Then it downloads a UTXO snapshot for some height from its peers. Finally, full blocks after the snapshot are to be downloaded and applied to get a current UTXO set. A pruned fullnode is also skipping AD-transformation block part, like a fullnode. Additional setting: "suffix" - how much full blocks to store(w. some minimum set?). Its regular modifiers processing is the same as for fullnode regime, while its bootstrap process is different:

1. Send ErgoSyncInfo message to connected peers.

2. Get response with INV message, containing ids of blocks, better than our best block.

3. Request headers for all ids from 2.

4. On receiving header:

   ```
   if(History.apply(header).isSuccess) {
       if(!(localScore == networkScore)) GOTO 1
       else GOTO 5
   } else {
       blacklist peer
   }
   ```

5. Request historical UTXOManifest for at least BlocksToKeep back.

6. On receiving UTXOSnapshotManifest:

   ```
   UTXOSnapshotManifest.chunks.foreach { chunk =>
       request chunk from sender() //Or from random fullnode
   }
   ```

7. On receiving UTXOSnapshotChunk:

   ```
   State.applyChunk(UTXOSnapshotChunk) match {
       case Success(Some(newMinimalState)) => GOTO 8
       case Success(None) => stay at 7
       /*we need more chunks to construct state.
       TODO periodicaly request missed chunks*/
       case Failure(e) => ???
       //UTXOSnapshotChunk or constcucted state roothash is invalid
   }
   ```

8. Request BlockTransactions starting from State we have

   ```
   History.headersStartingFromId(State.headerId).foreach { header =>
       send message(GetBlockTransactionsForHeader(header)) to Random fullnode
   }
   ```

9. On receiving BlockTransactions: same as in Fullnode.7 .

10. Operate as Fullnode.

Light Full-Node Mode

This mode is based on an idea to use a 2-party authenticated dynamic dictionary built on top of UTXO set. A light-fullnode holds only a root digest of a dictionary. It check all the full blocks, or some suffix of the full blockchain, depending on setting, thus starting from a trusted pre-genesis digest or some digest in the blockchain. A light-fullnode is using AD-transformations (authenticated dictionary transformations) block section containing batch-proof for UTXO transformations to get a new digest from an old one. It also checks all the transactions, but doesn't store anything but a single digest for that. Details can be found in the paper https://eprint.iacr.org/2016/994.

Additional settings : "depth" - from which block in the past to check transactions (if 0, then go from genesis).

"additional-checks" - light-fullnode trusts previous digest and checks current digest validity by using the previous one as well as AD-transformations.

"additional-depth" - depth to start additional checks from.

1. Send ErgoSyncInfo message to connected peers.

2. Get response with INV message, containing ids of blocks, better than our best block.

3. Request headers for all ids from 2.

4. On receiving header:

   ```
   if(History.apply(header).isSuccess) {
       if(!(localScore == networkScore)) GOTO 1
       else GOTO 5
    } else {
       blacklist peer
    }
   ```

5. Request BlockTransactions and ADProofs starting from BlocksToKeep back in History (just 1 last header after node botstrapping):

   ```
   History.lastBestHeaders(BlocksToKeep).foreach { header =>
       send message(GetBlockTransactionsForHeader(header)) to Random fullnode
       send message(GetAdProofsHeader(header)) to Random fullnode
     }
   ```

6. On receiving modifier BlockTransactions or ADProofs:

   ```
   if(History.apply(modifier) == Success(ProgressInfo)) {
     /* TODO if history now contains both ADProofs and BlockTransactions,
     it should return ProgressInfo with both of them, otherwise
     it should return empty ProgressInfo */
   if(State().apply(ProgressInfo) == Success((newState, ADProofs)))
   {
   if("mode"=="pruned-full") drop BlockTransactions and ADProofs older than BlocksToKeep
   }
   else {
           /*Drop Header from history, because it's transaction sequence is not valid*/
           History.drop(BlockTransactions.headerId)
       }
     }
   ```

Light-SPV Mode

This mode is not about checking any full blocks. Like in Bitcoin, an SPV node is downloading block headers only, and so checks only proofs of work and links. Unlike Bitcoin's SPV, the Light-SPV is downloading and

checking not all the headers but a sublinear(in blockchain length) number of them(in benchmarks, this is about just tens of kilobytes instead of tens or hundreds of megabytes for Bitcoin/Ethereum). Light-SPV mode is intended to be useful for mobile phones and low-end hardware.

Bootstrap

1. Send GetPoPoWProof for all connections.

2. On receive PoPoWProof apply it to History (History should be able to determine, whether this PoPoWProof is better, than it's current best header chain).

3. GOTO regular regime.

Regular

1. Send ErgoSyncInfo message to connected peers

2. Get response with INV message, containing ids of blocks, better than our best block.

3. Request headers for all ids from 2.

4. On receiving header:

```
if(History.apply(header).isSuccess) {
   State.apply(header) // just change state roothash
   if(!isInitialBootstrapping) Broadcast INV for this header
} else {
   blacklist peer
}
```

Mode-Related Settings

Ergo has the following settings determines a mode:

- ADState: Boolean - keeps state roothash only.

- VerifyTransactions: Boolean - download block transactions and verify them (requires BlocksToKeep == 0 if disabled).

- PoPoWBootstrap: Boolean - download PoPoW proof only

- BlocksToKeep: Int - number of last blocks to keep with transactions, for all other blocks it keep header only. Keep all blocks from genesis if negative

- MinimalSuffix: Int - minimal suffix size for PoPoW proof (may be pre-defined constant).

'if(VerifyTransactions == false) require(BlocksToKeep == 0)' Mode from **"multimode.md"** can be determined as follows:

Ergo Block Structure

ErgoMinimalHeader is a minimal data amount, required to calculate blockId:

payloadRootHash: Array[Byte] - root hash (or simple hash of all payload data) of block payload. nonce: Int - field to iterate and generate valid PoW.

ErgoHeader is a header to keep in History and transfer:

Field	Size	Description
version	1	block version, to be increased on every soft- and hardfork
parentId	32	id of parent block
interlinksRoot	32	root hash of interlinks structure
ADProofsRoot	32	hash of ADProofs for transactions in a block
stateRoot	32	root hash (for an AVL+ tree) of a state after block application
transactionsRoot	32	root hash (for a Merkle tree) of transactions in a block
timestamp	8	block timestamp(in milliseconds since beginning of Unix Epoch)
nonce	8	Proof-of-Work nonce

Some of this fields may be calculated by node by itself:

- parentId: if(status==bootstrap AND PoPoWBootstrap == false).

- interlinksRoot: if(PoPoWBootstrap == false).

- ADProofsRoot: if(status==regular AND ADState==false AND BlocksToKeep>0).

- stateRoot: if(status==regular AND ADState==false AND BlocksToKeep>0).

Ergo Modifiers Processing

This section describes processing algorithm for Ergo modifiers in all security modes. Unlike most of blockchain systems, Ergo have the following types of **modifiers**: In-memory:

1. In-memory:

   - Transaction - in-memory modifier.
   - TransactionIdsForHeader - ids of transactions in concrete block.
   - UTXOSnapshotManifest - ids of UTXO chunks and

2. Persistent:

   - BlockTransactions - Sequence of transactions, corresponding to 1 block.
   - ADProofs - proof of transaction correctness relative to corresponding UTXO.
   - Header, that contains data required to verify PoW, link to previous block, state root hash and root hash to it's payload (BlockTransactions, ADProofs, Interlinks ...).
   - UTXOSnapshotChunk - part of UTXO.
   - PoPoWProof

   Ergo will have the following parameters, that will determine concrete security regime:

   - ADState: Boolean - keep state roothash only.
   - VerifyTransactions: Boolean - download block transactions and verify them (requires BlocksToKeep == 0 if disabled).
   - PoPoWBootstrap: Boolean - download PoPoW proof only.
   - BlocksToKeep: Int - number of last blocks to keep with transactions, for all other blocks it keep header only. Keep all blocks from genesis if negative.

- MinimalSuffix: Int - minimal suffix size for PoPoW proof (may be pre-defined constant).

```
if(VerifyTransactions == false) require(BlocksToKeep == 0)
```

Mode from "multimode.md" can be determined as follows:

```
mode = if(ADState == false && VerifyTransactions == true
&& PoPoWBootstrap == false && BlocksToKeep < 0) "full"
else if(ADState == false && VerifyTransactions == true
&& PoPoWBootstrap == false && BlocksToKeep >= 0) "pruned-full"
else if(ADState == true && VerifyTransactions == true
&& PoPoWBootstrap == false) "light-full"
else if(ADState == true && VerifyTransactions == false
&& PoPoWBootstrap == true && BlocksToKeep == 0) "light-spv"
else if(ADState == true && VerifyTransactions == true
&& PoPoWBootstrap == true && BlocksToKeep == 0) "light-full-PoPoW"
else //Other combinations are possible
```

Modifiers processing

```
def updateHeadersChainToBestInNetwork() = {
  1.2.1. Send ErgoSyncInfo message to connected peers
  1.2.2. Get response with INV message,
  containing ids of blocks, better than our best block
  1.2.3. Request headers for all ids from 1.2.2.
  1.2.4. On receiving header
   if(History.apply(header).isSuccess) {
      if(!(localScore == networkScore)) GOTO 1.2.1
  } else {
      blacklist peer
      GOTO 1.2.1
  }
}
```

bootstrap

Download headers:

```
if(PoPoW) {
 1.1.1. Send GetPoPoWProof(suffix = Max(MinimalSuffix ,BlocksToKeep)) for all connections
 1.1.2. On receive PoPoWProof apply it to History
  /*
  History should be able to determine,
  whether this PoPoWProof is better, than it's current best header chain */
} else {
  updateHeadersChainToBestInNetwork()
}
```

Download initial State to start process transactions:

```
if(ADState == true) {
  Initialize state with state roothash from block header BlocksToKeep ago
} else if(BlocksToKeep < 0 || BlocksToKeep > History.headersHeight) {
  Initialize state with genesis State
} else {
/*
```

```
We need to download full state BlocksToKeep back in history
TODO what if we can download state only "BlocksToKeep - N"
or "BlocksToKeep + N" blocks back?
*/
  2.1. Request historical UTXOSnapshotManifest for at least BlocksToKeep back
  2.2. On receiving UTXOSnapshotManifest:
    UTXOSnapshotManifest.chunks.foreach ( chunk => request chunk from sender()
/*Or from random fullnode*/
  2.3. On receiving UTXOSnapshotChunk
  State.applyChunk(UTXOSnapshotChunk) match {
    case Success(Some(newMinimalState)) => GOTO 3
    case Success(None) => stay at 2.3
    /*we need more chunks to construct state. TODO periodicaly request missed chunks*/
    case Failure(e) => ???
    /*UTXOSnapshotChunk or constcucted state roothash is invalid*/
  }
}


Update State to best headers height:

 if(State.bestHeader == History.bestHeader) {
    //Do nothing, State is already updated
  } else if(VerifyTransactions == false) {
/*Just update State rootshash to best header in history*/
    State.setBestHeader(History.bestHeader)
  } else {
/*we have headers chain better then full block */
    3.1.
      assert(history contains header chain from State.bestHeader to History.bestHeaders)
      History.continuation(from = State.bestHeader, size = ???).get.foreach { header =>
        sendToRandomFullNode(GetBlockTransactionsForHeader(header))
        if(ADState == true) sendToRandomFullNode(GetADProofsForHeader(header))
      }
    3.2. On receiving modifiers ADProofs or BlockTransactions
      /*TODO History should return non-empty ProgressInfo
      only if it contains both ADProofs and BlockTransactions,
      or it contains BlockTransactions and ADState==false*/
      if(History.apply(modifier) == Success(ProgressInfo)) {
        if(State().apply(ProgressInfo) == Success((newState, ADProofs))) {
          if(ADState==false) ADProofs.foreach ( ADProof => History.apply(ADProof))
          if(BlocksToKeep>=0)
          /*remove BlockTransactions and ADProofs older than BlocksToKeep from history*/
        } else {
        /*Drop Header from history,
        because it's transaction sequence is not valid*/
          History.drop(modifier.headerId)
        }
      } else {
        blacklistPeer
      }
      GOTO 3
    }
```

GOTO regular mode.

Regular

Two infinite loops in different threads with the following functions inside:

1. UpdateHeadersChainToBestInNetwork()

2. Download and update full blocks when needed

```
if(State.bestHeader == History.bestHeader) {
  //Do nothing, State is already updated
} else if(VerifyTransactions == false) {
  //Just update State rootshash to best header in history
  State.setBestHeader(History.bestHeader)
} else {
  //we have headers chain better then full block
  3.1. Request transaction ids from all headers without transactions
    assert(history contains header chain from State.bestHeader to History.bestHeaders)
    History.continuation(from = State.bestHeader, size = ???).get.foreach { header =>
      sendToRandomFullNode(GetTransactionIdsForHeader(header))
      if(ADState == true) sendToRandomFullNode(GetADProofsForHeader(header))
    }
  3.2. On receiving TransactionIdsForHeader:
    Mempool.apply(TransactionIdsForHeader)
    TransactionIdsForHeader.filter(txId => !MemPool.contains(txId)).foreach { txId =>
      request transaction with txId
    }
  3.3. On receiving a transaction:
    if(Mempool.apply(transaction).isSuccess) {
      Broadcast INV for this transaction
      Mempool.getHeadersWithAllTransactions { BlockTransactions =>
        GOTO 3.4 //now we have BlockTransactions
      }
    }
  3.4. (same as 3.2. from bootstrap)
}
```

Components

History

State

Memory Pool

Transactional Language

Protocol Updates

Peer-to-Peer Network

Emission Rules

Alex notes : are these rules for money chain only?

First, we are describing some preliminaries before providing rules for coins emission. Target mean delay between blocks is $\Delta = 2$ minutes. We plan to issue all the tokens in about eight years, thus the number of blocks minting coins is $E_{total} = 365 \cdot 24 \cdot \frac{60}{\Delta} = 262800$.

Transactions

A transaction is destroying at least one coin (potentially many coins, up to $2^{16} - 1$ if block size limit allows that), and also creating at least one coin. We use the term "destroying", as everything contained in a coin is disappearing from the state during transaction application: monetary value, guard script, contents of all the registers, all the bytes and identifier of the coin.

Coin Format

A coin is made of registers, we allow every coin in the system to have up to 8 registers in the Σ-Cash chain, and up to 64 registers in the Σ-Data chain. We denote the registers as $R_0, R_1, \ldots, R_{63}$. From these registers, some are filled with mandatory values: R_0 contains monetary value of a coin, R_1 contains serialized guard script, R_2 contains unique identifier of transaction and also an index.

Registers are typed: every register contains a value of some type. Types are defined in Alex notes : ref. The value should be evaluated (i.e. be a constant value).

We introduce $extract()$ function, which is reading contents of a register, for example, $extract(c, R_0)$ extracts monetary value from the coin c.

Coin template

Here we describe difference between a coin and a coin template. A coin has a unique identifier to be defined deterministically from its contents. Thus we need to have different identifiers for coins of the same meaning, even if they are created by the same transaction. We also require a coin to have an identifier which is derived solely from coin contents, thus we can not use *(transaction_id, output_id)* pair as Bitcoin Core implementation is doing.

To solve the issue we split concepts of a coin and a coin template. A coin template is defining semantics of the corresponding coin i.e. has the same values for all the registers except of the register R_2 which is set to *null*.

Transaction Format

A transaction simply refers to a set of coins, by providing their identifiers and also authorization proofs for them; the transaction also providing set of new coin templates. A coin template becomes a coin in result of transaction processing.

We now introduce two functions to extract a coin or a coin template which a transaction is trying to spend. In details, function $in(index)$ returns a coin which transaction is trying to spend, by its index, and $out(index)$ returns a coin template . For example, $in(tx, 0)$ returns the very first coin the transaction tx is trying to spend.

We require for every transaction tx, which is trying to spend c_i coins and create c_o coins, to preserve overall monetary value:
$$\sum_{i=0}^{c_i-1} extract(in(tx, i), R_0) = \sum_{i=0}^{c_o-1} extract(out(tx, i), R_0)$$

Transaction Merkle Tree

Like a miner in the Bitcoin protocol is building a Merkle tree of block transactions, as well as a Merkle tree of transaction witnesses (after the Segwit upgrade), in Ergo, a miner should build a Merkle tree (and include a correct root hash of the tree into a block header), which is in case of Ergo combines both transactions and their respective spending proofs.

This tree is to be constructed as follows. A data block under a leaf of the tree could be empty or 64 bytes in length. Data of 64 bytes contains identifier of the transaction (256-bits digest of transaction bytes without spending proofs) and 256 bits of a digest of all the spending proofs for the transaction combined. Data for $i-th$ transaction in the block (starting from 0) is authenticated by the $i-th$ leaf. A leaf is $hash(0||pos||data)$, if the $data$ is not empty (we do add prefix for domain separation), or $null$ otherwise. Here, pos is a position of the transaction in the block. For internal nodes, a node is $hash(1||left\_child||right\_child)$, if either left child or right child of the node is not $null$, $null$ othewise. If root hash is $null$, we are writing all zeros (of hash function output length) instead of it.

Transaction Identifier and Its Malleability

A transaction has a unique identifier to be defined deterministically from its contents.

There are different ways to produce semantically the same transaction with different identifiers. We are interesting to prevent *unauthorized malleability*. We define unauthorized malleability as a way to change transaction identifier without re-signing the transaction. One particular way to malleate a transaction is to use signature malleability, which is common for most popular digital signature schemes.

To prevent transaction malleability, we do not put spending proofs under hash function when we calculate a transaction identifier.

Signing Transactions

Unified Transactions