The Ergo Yellowpaper

Alexander Chepurnoy Nazeem Faour Dmitry Meshkov

June 5, 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 skipping downloading and checking AD-transformation block part (see below in the "Light-Fullnode" section). For the full node regime, modifiers processing 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:
5. On receiving transaction ids from header:
6. On receiving a transaction:
7. Now we have BlockTransactions: all transactions corresponding to some 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:
5. Request historical UTXOManifest for at least BlocksToKeep back.
6. On receiving UTXOSnapshotManifest:
7. On receiving UTXOSnapshotChunk:
8. Request BlockTransactions starting from State we have
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 checks 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:
5. Request BlockTransactions and ADProofs starting from BlocksToKeep back in History (just 1 last header after node botstrapping):
6. On receiving modifier BlockTransactions or ADProofs:

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:

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).

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

Modifiers processing

bootstrap

Download headers:

Download initial State to start process transactions:

Update State to best headers height:

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

Components

History

State

Memory Pool

Transactional Language

Protocol Updates

Peer-to-Peer Network

Emission Rules

Ergo emission properties are defined by the following parameters in a monetary section of the config:

- *fixedRatePeriod* - number of blocks since the genesis block, the reward won't change. For mainnet it equals to 525600, that corresponds to 2 years of fixed rate (assuming 2 minutes block interval)
- *fixedRate* - number of coins issued every block during the $fixedRatePeriod$. For mainnet it will be set to 7500000000 (75 coins with 8 digits).
- *epochLength* - number of blocks between reward reduction. For mainnet it will be set to 64800 (3 month).
- *oneEpochReduction* - number of coins reward decrease every epochs. For mainnet it will be set to 300000000 (3 coins with 8 digits).

Thus for the first $fixedRatePeriod$ blocks block reward for miner will be fixed and equals to $fixedRate$ Ergo coins, after $fixedRatePeriod$ block reward will be reduced for *oneEpochReduction* coins every *epochLength* blocks.

Instead of having implicit emission rules via a special type of transaction (e.g. coinbase transaction in Bitcoin), Ergo coins emission is defined explicitly by $\Sigma-statetransactionallanguage.AllErgocoinswillbecreatedinthegenesisstate$

This protection script allows miner to take only a part of all coins, corresponding to current block reward. Fist, this script calculates $coinsToIssue$ - number of coins, current miner takes to himself, and checks, that the remaining number of coins in the first output *out* equals to initial number of coins minus $coinsToIssue$. This prevents the miner to take more coins, that is defined by emission rules. Second, script checks that register $R3$ of *out* contains current height and this height is greater, then height kept in $R3$ register of spending input. This prevents the miner to take use this output more then once per block. Finally, it checks that *out* have the same script, preventing miners to more coins then expected in the future. Special case is required to stop emission - when this output contains less coins, than *oneEpochReduction*, creation of a new output is not required.

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 also an identifier of the coin.

Coin Format

A coin is made of registers (and nothing but 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 which created the coin and also an index of the coin in the transaction.

Each register is an expression in sigmalanguage. Thus the 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 bytes to be used to get the coin identifier, build authenticated tree for the state, and build a transaction, are to be formed as follows:

- *monetary value.* We use 64 bits long *signed* integer value to store monetary value of the coin (we use the signed integer to make processing easier on platforms with no unsigned integers, such as JVM platform). This value to be represented as register R0 by wrapping it as a constant of integer type.
- *bytes of a script.* To see how the script is serialized, see (Alex notes : link to sigmalanguage expressions serialization). The script is to be represented as register R1 by wrapping its bytes are constant array of constant bytes.
- *number of additional registers.* Extra registers should come in order (R3, R4, ..., etc), so this number, represented as 1-byte value, defines how much non-empty registers are coming after mandatory ones. If the number is zero, next field is missed.
- *additional registers.* Extra registers are serialized without any delimiters. Each register is serialized as a sigmaexpression.
- *transaction identifier and index of a transaction output.* Identifier of a transaction which created the coin, 32-bytes long, as well as index of the coinin the transaction outputs, 2-bytes long. These values are to be concatenated into a single array of 34 bytes and wrapped into sigmalanguage constant stored in the register $R2$.

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 spending proofs for them; the transaction also providing set of new coin templates. A coin template becomes a coin in result of transaction processing (however, once transaction is formed, its identifier is known, and so identifier of the future coin is also known before the application).

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:
$\sum_{i=0}^{c_i-1} extract(in(tx, i), R_0) = \sum_{j=0}^{c_o-1} extract(out(tx, j), R_0)$

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

In details, we are constructing bytes to get the transaction identifier (and in the follow-up section, also a signature) as follows. We concatenate the following fields:

- *number of inputs.* Unsigned 2-bytes integer giving a number of coins the transaction is spending. Thus the transaction can not spend more than 65536 coins.
- *input identifiers.* Given the number c_{ins} of coins to spend from the previous field, we form $32 * c_{ins}$ bytes, where i-th chunk of 32 bytes represents an identifier of i-th coin to spend (see Section **??** for details).
- *number of outputs.* Unsigned 2-bytes integer of coins the transaction is creating (again, the transaction can not create more than 65,536 coins).
- *transaction outputs.* In this section we store serialized coin templates, with no delimiters. A coin template is serialized as a coin (see Section **??** for details), but without last field (transaction identifier and index of the coin in transaction outputs).

Denoting the binary serialization procedure for a transaction tx explained above as $bytes(tx)$, we can derive transaction identifier $ident(tx)$ from it as:

$$ident(tx) = hash(bytes(tx))$$

Signing A Transaction

To spend a coin a spending transaction tx has as an input, one needs to use $bytes(tx)$ (note that different inputs can be signed in parallel, however, the coins being spent are to be specified before signing), as well as current state of the blockchain, or *context*. To avoid any I/O operations, this context is small. In details, it consists of current height, last block's state root digest, and the spending transaction along with coins it spends. The signer also can extend the context by putting values there.

By having this data, a signer (or a prover) of an input first reduces the guarding proposition for the input coin by evaluating, it using the context. Possible results of the reduction are:

- abort if estimated upper-bound cost of evaluation (and verification) is too high.
- true means that the coin is spendable by anyone
- false means that the coin is not spendable at all (at least according to the current context)
- expression still containing predicates over the context. That means context is not enough to evaluate some predicates over it. Prover can look into its own not revealed yet secrets in order to extend context. If the secrets are found, prover is reducing the expression further and doing the next step, if there is nothing more to evaluate. Otherwise the prover aborts.
- expression containing only expressions over secret information provable via Σ-protocols. This is the most common case, and we are going to explain it in details further.

We are having possible complex expression, like $dlog(x_1) \vee (dlog(x2) / \ dlog(x3))$, where $dlog(x)$ means "prove me knowledge of a secret w, such as for a known group with generator g, $g^w = x$, via a non-interactive form of the Schnorr protocol". Internally, this exression is represented as a tree (Alex notes : draw the tree). This proof is to be proven and then serialized into a binary spending proof (which could be included into a transaction) as follows:

Proving steps for a tree:

Alex notes : text below is duplicated in Sigma paper, also this is a raw text copypasted from an email

1. bottom-up: mark every node real or simulated, according to the following rule. DLogNode – if you know the secret, then real, else simulated. $\vee$: if at least one child real, then real; else simulated. $\wedge$: if at least one child simulated, then simulated; else real. Note that all descendants of a simulated node will be later simulated, even if they were marked as real. This is what the next step will do.

Root should end up real according to this rule – else you won't be able to carry out the proof in the end.

2. top-down: mark every child of a simulated node "simulated." If two or more more children of a real $\vee$ are real, mark all but one simulated.

3. top-down: compute a challenge for every simulated child of every $\vee$ and $\wedge$, according to the following rules. If $\vee$, then every simulated child gets a fresh random challenge. If $\wedge$ (which means $\wedge$ itself is simulated, and all its children are), then every child gets the same challenge as the $\wedge$.

4. bottom-up: For every simulated leaf, simulate a response and a commitment (i.e., second and first prover message) according to the Schnorr simulator. For every real leaf, compute the commitment (i.e., first prover message) according to the Schnorr protocol. For every $\vee/\wedge$ node, let the commitment be the union (as a set) of commitments below it.

5. Compute the Schnorr challenge as the hash of the commitment of the root (plus other inputs – probably the tree being proven and the message).

6. top-down: compute the challenge for every real child of every real $\vee$ and $\wedge$, as follows. If $\vee$, then the challenge for the one real child of $\vee$ is equal to the XOR of the challenge of $\vee$ and the challenges for all the simulated children of $\vee$. If $\wedge$, then the challenge for every real child of $\wedge$ is equal to the the challenge of the $\wedge$. Note that simulated $\wedge$ and $\vee$ have only simulated descendants, so no need to recurse down from them.

Now, how to get a flat binary string containing (e, z) pairs (challenge and prover's response for a leafsub-protocol) from the tree:

Verifying A Transaction

Unified Transactions