Trustless interaction with TON Blockchain

by RSquad Blockchain Lab on behalf of TON Foundation

v1.1, 15 May 2023

Contents

1 Getting data

During operation, the system may need to have:

- block, transaction or proof (boc format);
- validator signatures for a specific block.

The source of data for the system are:

- proofs — get through the `toncenter`;
- blocks — get via `lite api` or `lite client`;
- signatures — receive through the `toncenter`.

2 Initialization

To start the trustless system, you must first prepare:

- a list of validators (their pubkeys) from the initial key block, from which it will be possible to start checking the next blocks (including key blocks with a new list of validators);
- root hash of the key block with "initial" validators can also be stored in the system during initialization (optionally, the root hash can be used to validate old blocks, but root hashes of other blocks can be used for the same purpose, which will later be stored in the system during validation).

For initialization, we recommend to use either an up-to-date list of validators, or a list of validators from relatively new keyblocks, since for old blocks there may not be signatures saved.

2.1 Algorithm

1. Get a key block with a list of validators.

```
// (lite_api) POST /v1/lite_server_get_block

{
  "id": {
    "workchain": -1,
    "shard": "-9223372036854775808",
    "seqno": 6142808,
    "root_hash": "f241...a157",
    "file_hash": "69ed...3fc2"
  }
}
```

2. Read information about validators from the block. Path to the list of validators in the block according to `block.tlb` — `block -> extra -> custom -> config -> configParam34`.
3. Save necessary data — `node_id`, `pubkey` of validators, (optionally) save the `root_hash` of the block.

2.2 Example

```
// Used 'ton', modified 'ton-core' and utils for
// read boc (based on 'ton-core')

const initialKeyBlockWithShards = await tonClient4.getBlock(
  initKeyBlockSeqno
);

const initialKeyBlockInformation = initialKeyBlockWithShards
  .shards
```

```
  .find(
    (blockRes) => blockRes.seqno === initKeyBlockSeqno
  );

if (!initialKeyBlockInformation) {
  throw Error("Block not found");
}

const initialKeyBlockWithBase64 =
  await getBlock(initialKeyBlockInformation);
const initialKeyBlockRootCell = readBlockRootCell(
  initialKeyBlockWithBase64.data,
  initialKeyBlockWithBase64.id.root_hash
);

const initialKeyBlock = loadBlock(initialKeyBlockRootCell);

const config: Dictionary<number, Cell> =
  initialKeyBlock.extra.custom?.config!;
const validatorsSetCell = config.get(34);

if (!validatorsSetCell) {
  throw Error(`
    Block isn't keyblock or block has no new validator set
  `);
}

const validators = loadValidatorSet(validatorsSetCell);
const list = validators.list.values();
await trustlessState.updateValidators(
  list,
  validators.total_weight || 0n
);
```

3 Proofs check

All proofs, such as proof of commiting a shard block into a masterchain block or proof of state, are checked for correctness by checking their structure - by counting the hashes of cells and by checking the hash of the root cell (its hash must be either hash of a verified block, or a verified hash of a state).

To read data from a proof or block, you need to implement bag of cells parser according to the data structure from `block.tlb`.

If there is no proof and the block is a masterchain block, then it can be verified by checking the signatures of validators from the key block of the same epoch. To do this, using the Ed25519 algorithm, the public keys of the validators and the signature data of these validators on the submitted block are checked (to check 1 signature of the validator `public key`, `r + s signatures`, `root_hash` and `file_hash` are needed). To get signatures, you can use the toncenter method `GET /getMasterchainBlockSignatures`.

Blocks verified by the system are saved as their `root_hash` for further use in validating new blocks.

In total, to use various checks, the system must store (cache) the following data:

- hashes of validated blocks and `state_hash` corresponding to these blocks for checks using proofs. With them, it is checked that the submitted proof is correct (the calculated hash of the proof will match the hash of the block or, in the case of the proof of the state, the hash of the block state)
- a list of validators of the current epoch, taken from the last keyblock to verify new masterchain blocks using signatures.

4 How to sync masterchain

For minimal masterchain synchronization in a system using trustless interaction, it is necessary to verify incoming masterchain key blocks and, if they contain a new list of validators, update the list of validators in the system. Without this condition, verification by the signatures of validators will not work, because it is impossible to find out the current list of validators.

It is also possible to validate each incoming block from the TON by the trustless system and achieve full synchronization, but, in most cases, it is not necessary.

4.1 By keyblocks

One way to validate a block from the masterchain is to proof it to a **newer** masterchain block (which has already been verified by the trustless system).

Prerquisites:

- the system stores the hash of the verified block of the masterchain and the `state_hash` for this block (`state_hash` is the `new_hash` taken from the block along the path `block -> state_update -> new_hash`);
- the block we want to check is **older** than the already checked block.

4.1.1 Algorithm

1. Obtain a proof to the block we want to check from the masterchain block

```
// (lite_api) POST /v1/lite_server_get_block_proof

{
  "id": {
    "workchain": -1,
    "shard": "-9223372036854775808",
    "seqno": 6142808,
    "root_hash": "f241...a157",
    "file_hash": "69ed...3fc2"
  },
  "target_block": null
}
```

2. Read information from the proof along the way —
   `shard_state -> custom -> prev_blocks -> blk_ref (ExtBlkRef)`
3. Check if hash of `block_proof`
   (`boc_proof_root_cell.refs[0].hashes()[0]`)
   and state hash of masterchain block
   (`block -> state_update -> new_hash`) are equal

As a result, we got a list of structures with `root_hash` and `file_hash` blocks going up to `target_block`. All blocks from `blk_ref` contains valid `root_hashes` of masterchain blocks (including the `root_hash` of the block being checked).

4.1.2 Example

```
// Used 'ton', modified 'ton-core' and utils for
// read boc (based on 'ton-core')

// Verify block by state_hash
const mc_proof = (
  await axios.get(
    toncenterUrl +
      `getShardBlockProof?
        workchain=${initialKeyBlockInfo.id.workchain}&
        shard=${initialKeyBlockInfo.id.shard}&
        seqno=${initialKeyBlockInfo.id.seqno}&
        from_seqno=${newKeyBlockSeqno}`
  )
).data.result.mc_proof[0];

const mcProofRootCell = readBlockRootCell(mc_proof.state_proof);

const shardStateUnsplit = loadShardStateUnsplit(
  mcProofRootCell.beginParse(true).loadRef().beginParse()
);
const validBlocks =
  shardStateUnsplit.extras?.prev_blocks
    ?.values()
    .map((v) => v.value.blk_ref) || [];

if (
  Buffer.from(mcProofRootCell.refs[0].hash(0)).toString("hex") !==
  trustlessState
    .verifiedBlocks[newKeyBlockInfo.id.root_hash]
    .state_hash
) {
  throw Error("wrong state proof");
}

validBlocks.forEach((blk_ref) => {
  if (!trustlessState.verifiedBlocks[blk_ref.root_hash]) {
    trustlessState.verifiedBlocks[blk_ref.root_hash] = {
      verified: true,
    };
  }
});
```

4.2 By signatures of validators of the current epoch

4.2.1 Prerequisites

- system stores a list of validators for the epoch in which the masterchain block was created.

4.2.2 Algorithm

1. Get a block in the boc format

```
// (lite_api) POST /v1/lite_server_get_block.

{
  "id": {
    "workchain": -1,
    "shard": "-9223372036854775808",
    "seqno": 6142808,
    "root_hash": "f241...a157",
    "file_hash": "69ed...3fc2"
  }
}
```

2. Get a list of signatures
   `GET /v2/getMasterchainBlockSignatures?seqno=6142808`
3. Check validator signatures (using Ed25519)
4. Check the condition — the block was signed by at least $\frac{2}{3}$ (by weight) of the total number of validators
5. According to the `block.tlb`, the list of new validators in the keyblock is stored in the following path (after reading data from boc) —
   `block -> extra -> custom -> config -> configParam34`

4.2.3 Example

```
// Used 'ton', modified 'ton-core' and utils for
// read boc (based on 'ton-core')

const newKeyBlockWithShards = await tonClient4.getBlock(
  newKeyBlockSeqno
);
const newKeyBlockInformation = newKeyBlockWithShards.shards.find(
  (blockRes) => blockRes.seqno === newKeyBlockSeqno
);

if (!newKeyBlockInformation) {
  throw Error("Block not found");
}

const newKeyBlockWithBase64 = await getBlock(newKeyBlockInformation);
const newKeyBlockRootCell = readBlockRootCell(
  newKeyBlockWithBase64.data,
  newKeyBlockWithBase64.id.root_hash
);

const newKeyBlock = loadBlock(newKeyBlockRootCell);

let total_weight = new BN(trustlessState.totalWeight.toString());
let signed_weight = new BN(0);

// cannot get from ton client
const signatures: Signature[] = (
  await axios.get(
    `${toncenterUrl}getMasterchainBlockSignatures?
```

```
      seqno=${newKeyBlockWithBase64.id.seqno}`
  )
).data.result.signatures;

for (let j = 0; j < signatures.length; j++) {
  const signature = signatures[j];
  const s = Buffer.from(signature.node_id_short, "base64")
    .toString("hex");

  const vs = trustlessState.validators;
  const i = vs.findIndex((v) => v.node_id === s);
  const validator = vs[i];

  let to_sign = new Uint8Array(68);
  to_sign.set([0x70, 0x6e, 0x0b, 0xc5], 0);
  to_sign.set(
    new Uint8Array(Buffer.from(
      newKeyBlockWithBase64.id.root_hash,
      "hex"
    )),
    4
  );
  to_sign.set(
    new Uint8Array(Buffer.from(
      newKeyBlockWithBase64.id.file_hash,
      "hex"
    )),
    36
  );

  const res = signVerify(
    Buffer.from(to_sign),
    Buffer.from(signature.signature, "base64"),
    Buffer.from(validator.public_key, "hex")
  );

  if (!res) throw Error("signature check failed");

  signed_weight.iadd(new BN(validator.weight));

  if (signed_weight.gt(total_weight)) {
    break;
  }
}

signed_weight.imul(new BN(3));
total_weight.imul(new BN(2));

if (signed_weight.lte(total_weight))
  throw Error("insufficient total signature weight");

trustlessState
  .verifiedBlocks[newKeyBlockWithBase64.id.root_hash] = {
    verified: true,
    state_hash: newKeyBlock.state_update.new_hash,
};
```

```
if (newKeyBlock.info.key_block) {
  const config:
    Dictionary<number, Cell> = newKeyBlock.extra.custom?.config!;
  const validatorsSetCell = config.get(34);

  if (!validatorsSetCell) {
    throw Error(‘
      Block isn’t keyblock or block has no new validator set
    ‘);
  }

  const validators = loadValidatorSet(validatorsSetCell);
  const list = validators.list.values();
  await trustlessState.updateValidators(
    list, validators.total_weight || 0n
  );
}
```

5 How to verify a block

5.1 If block is a masterchain block

This scenario is described in the section above.

5.2 If block is a shardchain block

If a block is a shardchain block its enought to check this block commited to masterchain.

If the masterchain block from which we take the proof is verified by a trustless system, then all hashes from the `shard_hashes` of the proof are hashes of shardblocks from the blockchain.

5.2.1 Algorithm

1. Get a shard proof

```
// (lite_api) POST v1/lite_server_get_shard_block_proof

{
  "id": {
    "workchain": 0,
    "shard": "-9223372036854775808",
    "seqno": 5456001,
    "root_hash": "DC98...F451",
    "file_hash": "7263...6984"
  }
}
```

2. The path in the proof to the list of shardblock hashes that are committed to the masterchain block —
   `block -> extra -> custom -> shard_hashes`

3. Check if hash of root cell of shard proof
   (`boc_shard_proof.refs[0].hashes()[0]`)
   is equal to masterchain block `root_hash`

As a result, we will get a list of `root_hash` stored in `shard_hashes`, which are the hashes of the shard blocks that were committed to this masterchain block.

5.2.2 Example

```
// Used 'ton', modified 'ton-core' and utils for
// read boc (based on 'ton-core')

// use toncenter request instead of liteapi -
// liteapi's request has no needed field
const shardProofRes = (
  await axios.get(
```

```
    toncenterUrl +
      `getShardBlockProof?
        workchain=${someShard.workchain}&
        shard=${someShard.shard}&
        seqno=${someShard.seqno}&
        from_seqno=${block.seqno}`
  )
).data.result;

const shartProofRootCell =
  readBlockRootCell(shardProofRes.links[0].proof);

if (
  !trustlessState.verifiedBlocks[
    Buffer.from(shartProofRootCell.refs[0].hash(0)).toString("hex")
  ].verified
) {
  throw Error("verify ms block or use correct shard proof");
}

const shardProofData = loadBlock(shartProofRootCell.refs[0]);
const shardHashes = shardProofData.extra.custom!.shard_hashes;
shardHashes.values().forEach((binTreeItem) => {
  // type can be 'leaf' or 'fork'.
  // 'fork' has left/right fields contains a 'fork' or 'leaf' type.
  // in example we check only 'leaf' type
  if (binTreeItem.type === "leaf") {
    trustlessState.verifiedBlocks[binTreeItem.leaf!.root_hash] = {
      verified: true,
    };
  }
});
```

6 How to verify a transaction

6.1 Prerequisites

- The block with the transaction has already been verified by the trustless system by one of the methods and is a blockchain block.
- We know the hash of the transaction or we have the boc of the transaction. In the last case, we calculate the hash of the transaction using its boc.

6.2 Algorithm

1. Getting block as boc via lite_api

   ```
   // (lite_api) POST /v1/lite_server_get_block.

   {
     "id": {
       "workchain": -1,
       "shard": "-9223372036854775808",
       "seqno": 6142808,
       "root_hash": "f241...a157",
       "file_hash": "69ed...3fc2"
     }
   }
   ```

   (if the system is not limited by resources, you can immediately take a block with a transaction and then you do not need to check its entry into the block)

2. Read list of transactions from boc. The path to transactions in the block (you can use another one, but at the same time we check the match of the account in `account_block` and in the transaction) — `block -> extra -> account_blocks -> hashmap(ShardAccountBlocks) -> AccountBlock -> transactions`. If we don't know needed account, we can check every `account_block` in a loop

3. Find a transansaction in block by checking hash of transaction

4. If we found our transaction in block, transaction is verified. Now we can operate with transaction data (get messages from `transaction -> in_msg` or `transaction -> out_msgs` and get message body `message -> body`)

As a result, we will get a list of `root_hash` stored in `shard_hashes`, which are the hashes of the shard blocks that were committed to this masterchain block.

6.3 Example

```
// Used 'ton', modified 'ton-core' and utils for
// read boc (based on 'ton-core')

// In this example we get some transaction from
//verified blocks (if it exists), in real case you
// know transaction hash, lt and(optional) transaction address

const verifiedBlocksSeqno = [initKeyBlockSeqno, newKeyBlockSeqno];

for (let i = 0; i < verifiedBlocksSeqno.length; i++) {
  const shards = await tonClient4.getBlock(verifiedBlocksSeqno[i]);
  const verifiedBlock = shards.shards.find(
    (shard) => shard.seqno === verifiedBlocksSeqno[i]
  );
  if (!verifiedBlock) {
    throw Error("Unexpected result");
  }
  const transactions = verifiedBlock.transactions.map((tr) => {
    const accBuf = Buffer.from(tr.account, "base64");

    return {
      ...tr,
      accountAddr: accBuf
        .slice(2, accBuf.length - 2)
        .toString("hex"),
      hash: Buffer.from(tr.hash, "base64").toString("hex"),
    };
  });

  if (!transactions.length) {
    continue;
  }

  const transactionForCheck = transactions[0];

  const varifiedBlockInfo = await getBlock(verifiedBlock);
  const verifiedBlockRootCell = readBlockRootCell(
    varifiedBlockInfo.data,
    varifiedBlockInfo.id.root_hash
  );
  const verifiedBlockData = loadBlock(verifiedBlockRootCell);
  for (let [accountAddr, transactions] of verifiedBlockData.extra
    .account_blocks!) {
    if (transactionForCheck.accountAddr ===
      accountAddr.toString("hex")) {
      const transactionsOfAddr =
        transactions.value.transactions.values();
      const transaction = transactionsOfAddr.find(
        (tr) =>
          tr.value.lt.toString() === transactionForCheck.lt &&
          tr.value.hash === transactionForCheck.hash
      );
      console.log(transaction?.value);
      break;
    }
```

```
  }
}
```