Telegram Open Network

Dr. Nikolai Durov

August 12, 2023

Abstract

The aim of this text is to provide a first description of the Telegram Open Network (TON) and related blockchain, peer-to-peer, distributed storage and service hosting technologies. To reduce the size of this document to reasonable proportions, we focus mainly on the unique and defining features of the TON platform that are important for it to achieve its stated goals.

Introduction

The *Telegram Open Network (TON)* is a fast, secure and scalable blockchain and network project, capable of handling millions of transactions per second if necessary, and both user-friendly and service provider-friendly. We aim for it to be able to host all reasonable applications currently proposed and conceived. One might think about TON as a huge distributed supercomputer, or rather a huge "superserver", intended to host and provide a variety of services.

This text is not intended to be the ultimate reference with respect to all implementation details. Some particulars are likely to change during the development and testing phases.

Contents

1 Brief Description of TON Components

The *Telegram Open Network (TON)* is a combination of the following components:

- A flexible multi-blockchain platform (*TON Blockchain*; cf. Chapter **2**), capable of processing millions of transactions per second, with Turing-complete smart contracts, upgradable formal blockchain specifications, multi-cryptocurrency value transfer, support for micropayment channels and off-chain payment networks. *TON Blockchain* presents some new and unique features, such as the "self-healing" vertical blockchain mechanism (cf. **2.1.17**) and Instant Hypercube Routing (cf. **2.4.20**), which enable it to be fast, reliable, scalable and self-consistent at the same time.

- A peer-to-peer network (*TON P2P Network*, or just *TON Network*; cf. Chapter **3**), used for accessing the TON Blockchain, sending transaction candidates, and receiving updates about only those parts of the blockchain a client is interested in (e.g., those related to the client's accounts and smart contracts), but also able to support arbitrary distributed services, blockchain-related or not.

- A distributed file storage technology *(TON Storage;* cf. **4.1.8**), accessible through *TON Network*, used by the TON Blockchain to store archive copies of blocks and status data (snapshots), but also available for storing arbitrary files for users or other services running on the platform, with torrent-like access technology.

- A network proxy/anonymizer layer *(TON Proxy;* cf. **4.1.11** and **3.1.6**), similar to the I^2P (Invisible Internet Project), used to hide the identity and IP addresses of *TON Network* nodes if necessary (e.g., nodes committing transactions from accounts with large amounts of cryptocurrency, or high-stake blockchain validator nodes who wish to hide their exact IP address and geographical location as a measure against DDoS attacks).

- A Kademlia-like distributed hash table (*TON DHT*; cf. **3.2**), used as a "torrent tracker" for *TON Storage* (cf. **3.2.10**), as an "input tunnel locator" for *TON Proxy* (cf. **3.2.14**), and as a service locator for *TON Services* (cf. **3.2.12**).

- A platform for arbitrary services (*TON Services*; cf. Chapter **4**), residing in and available through *TON Network* and *TON Proxy*, with formalized interfaces (cf. **4.3.14**) enabling browser-like or smartphone application interaction. These formal interfaces and persistent service entry points can be published in the TON Blockchain (cf. **4.3.17**); actual nodes providing service at any given moment can be looked up through the *TON DHT* starting from information published in the TON Blockchain (cf. **3.2.12**). Services may create smart contracts in the TON Blockchain to offer some guarantees to their clients (cf. **4.1.7**).

- *TON DNS* (cf. **4.3.1**), a service for assigning human-readable names to accounts, smart contracts, services and network nodes.

- *TON Payments* (cf. Chapter **5**), a platform for micropayments, micropayment channels and a micropayment channel network. It can be used for fast off-chain value transfers, and for paying for services powered by *TON Services*.

- TON will allow easy integration with third-party messaging and social networking applications, thus making blockchain technologies and distributed services finally available and accessible to ordinary users (cf. **4.3.24**), rather than just to a handful of early cryptocurrency adopters. We will provide an example of such an integration in another of our projects, the Telegram Messenger (cf. **4.3.19**).

While the TON Blockchain is the core of the TON project, and the other components might be considered as playing a supportive role for the blockchain, they turn out to have useful and interesting functionality by themselves. Combined, they allow the platform to host more versatile applications than it would be possible by just using the TON Blockchain (cf. **2.9.13** and **4.1**).

2 TON Blockchain

We start with a description of the Telegram Open Network (TON) Blockchain, the core component of the project. Our approach here is "top-down": we give a general description of the whole first, and then provide more detail on each component.

For simplicity, we speak here about *the* TON Blockchain, even though in principle several instances of this blockchain protocol may be running independently (for example, as a result of hard forks). We consider only one of them.

2.1 TON Blockchain as a Collection of 2-Blockchains

The TON Blockchain is actually a *collection* of blockchains (even a collection of *blockchains of blockchains*, or *2-blockchains*—this point will be clarified later in **2.1.17**), because no single blockchain project is capable of achieving our goal of processing millions of transactions per second, as opposed to the now-standard dozens of transactions per second.

2.1.1. List of blockchain types. The blockchains in this collection are:

- The unique *master blockchain*, or *masterchain* for short, containing general information about the protocol and the current values of its parameters, the set of validators and their stakes, the set of currently active workchains and their "shards", and, most importantly, the set of hashes of the most recent blocks of all workchains and shardchains.
- Several (up to 2^{32}) *working blockchains*, or *workchains* for short, which are actually the "workhorses", containing the value-transfer and smart-contract transactions. Different workchains may have different "rules", meaning different formats of account addresses, different formats of transactions, different virtual machines (VMs) for smart contracts, different basic cryptocurrencies and so on. However, they all must satisfy certain basic interoperability criteria to make interaction between different workchains possible and relatively simple. In this respect, the TON Blockchain is *heterogeneous* (cf. **2.8.8**), similarly to the EOS (cf. **2.9.7**) and PolkaDot (cf. **2.9.8**) projects.
- Each workchain is in turn subdivided into up to 2^{60} *shard blockchains*, or *shardchains* for short, having the same rules and block format as

the workchain itself, but responsible only for a subset of accounts, depending on several first (most significant) bits of the account address. In other words, a form of sharding is built into the system (cf. **2.8.12**). Because all these shardchains share a common block format and rules, the TON Blockchain is *homogeneous* in this respect (cf. **2.8.8**), similarly to what has been discussed in one of Ethereum scaling proposals.[1]

- Each block in a shardchain (and in the masterchain) is actually not just a block, but a small blockchain. Normally, this "block blockchain" or "vertical blockchain" consists of exactly one block, and then we might think this is just the corresponding block of the shardchain (also called "horizontal blockchain" in this situation). However, if it becomes necessary to fix incorrect shardchain blocks, a new block is committed into the "vertical blockchain", containing either the replacement for the invalid "horizontal blockchain" block, or a "block difference", containing only a description of those parts of the previous version of this block that need to be changed. This is a TON-specific mechanism to replace detected invalid blocks without making a true fork of all shardchains involved; it will be explained in more detail in **2.1.17**. For now, we just remark that each shardchain (and the masterchain) is not a conventional blockchain, but a *blockchain of blockchains*, or *2D-blockchain*, or just a *2-blockchain*.

2.1.2. Infinite Sharding Paradigm. Almost all blockchain sharding proposals are "top-down": one first imagines a single blockchain, and then discusses how to split it into several interacting shardchains to improve performance and achieve scalability.

The TON approach to sharding is "bottom-up", explained as follows.

Imagine that sharding has been taken to its extreme, so that exactly one account or smart contract remains in each shardchain. Then we have a huge number of "account-chains", each describing the state and state transitions of only one account, and sending value-bearing messages to each other to transfer value and information.

Of course, it is impractical to have hundreds of millions of blockchains, with updates (i.e., new blocks) usually appearing quite rarely in each of them. In order to implement them more efficiently, we group these "account-chains" into "shardchains", so that each block of the shardchain is essentially

[1] `https://github.com/ethereum/wiki/wiki/Sharding-FAQ`

a collection of blocks of account-chains that have been assigned to this shard. Thus the "account-chains" have only a purely virtual or logical existence inside the "shardchains".

We call this perspective the *Infinite Sharding Paradigm*. It explains many of the design decisions for the TON Blockchain.

2.1.3. Messages. Instant Hypercube Routing. The Infinite Sharding Paradigm instructs us to regard each account (or smart contract) as if it were in its own shardchain by itself. Then the only way one account might affect the state of another is by sending a *message* to it (this is a special instance of the so-called Actor model, with accounts as Actors; cf. **2.4.2**). Therefore, a system of messages between accounts (and shardchains, because the source and destination accounts are, generally speaking, located in different shardchains) is of paramount importance to a scalable system such as the TON Blockchain. In fact, a novel feature of the TON Blockchain, called *Instant Hypercube Routing* (cf. **2.4.20**), enables it to deliver and process a message created in a block of one shardchain into the very next block of the destination shardchain, *regardless of the total number of shardchains in the system.*

2.1.4. Quantity of masterchains, workchains and shardchains. A TON Blockchain contains exactly one masterchain. However, the system can potentially accommodate up to 2^{32} workchains, each subdivided into up to 2^{60} shardchains.

2.1.5. Workchains can be virtual blockchains, not true blockchains. Because a workchain is usually subdivided into shardchains, the existence of the workchain is "virtual", meaning that it is not a true blockchain in the sense of the general definition provided in **2.2.1** below, but just a collection of shardchains. When only one shardchain corresponds to a workchain, this unique shardchain may be identified with the workchain, which in this case becomes a "true" blockchain, at least for some time, thus gaining a superficial similarity to customary single-blockchain design. However, the Infinite Sharding Paradigm (cf. **2.1.2**) tells us that this similarity is indeed superficial: it is just a coincidence that the potentially huge number of "account-chains" can temporarily be grouped into one blockchain.

2.1.6. Identification of workchains. Each workchain is identified by its *number* or *workchain identifier* ($workchain\_id : uint_{32}$), which is simply an

unsigned 32-bit integer. Workchains are created by special transactions in the masterchain, defining the (previously unused) workchain identifier and the formal description of the workchain, sufficient at least for the interaction of this workchain with other workchains and for superficial verification of this workchain's blocks.

2.1.7. Creation and activation of new workchains. The creation of a new workchain may be initiated by essentially any member of the community, ready to pay the (high) masterchain transaction fees required to publish the formal specification of a new workchain. However, in order for the new workchain to become active, a two-thirds consensus of validators is required, because they will need to upgrade their software to process blocks of the new workchain, and signal their readiness to work with the new workchain by special masterchain transactions. The party interested in the activation of the new workchain might provide some incentive for the validators to support the new workchain by means of some rewards distributed by a smart contract.

2.1.8. Identification of shardchains. Each shardchain is identified by a couple $(w, s) = (\mathit{workchain\_id}, \mathit{shard\_prefix})$, where $\mathit{workchain\_id} : \mathit{uint}_{32}$ identifies the corresponding workchain, and $\mathit{shard\_prefix} : \mathbf{2}^{0\ldots60}$ is a bit string of length at most 60, defining the subset of accounts for which this shardchain is responsible. Namely, all accounts with *account_id* starting with *shard_prefix* (i.e., having *shard_prefix* as most significant bits) will be assigned to this shardchain.

2.1.9. Identification of account-chains. Recall that account-chains have only a virtual existence (cf. **2.1.2**). However, they have a natural identifier—namely, $(\mathit{workchain\_id}, \mathit{account\_id})$—because any account-chain contains information about the state and updates of exactly one account (either a simple account or smart contract—the distinction is unimportant here).

2.1.10. Dynamic splitting and merging of shardchains; cf. 2.7. A less sophisticated system might use *static sharding*—for example, by using the top eight bits of the *account_id* to select one of 256 pre-defined shards.

An important feature of the TON Blockchain is that it implements *dynamic sharding*, meaning that the number of shards is not fixed. Instead, shard (w, s) can be automatically subdivided into shards $(w, s.0)$ and $(w, s.1)$ if some formal conditions are met (essentially, if the transaction load on the original shard is high enough for a prolonged period of time). Conversely,

if the load stays too low for some period of time, the shards $(w, s.0)$ and $(w, s.1)$ can be automatically merged back into shard (w, s).

Initially, only one shard $(w, \emptyset)$ is created for workchain w. Later, it is subdivided into more shards, if and when this becomes necessary (cf. **2.7.6** and **2.7.8**).

2.1.11. Basic workchain or Workchain Zero. While up to 2^{32} workchains can be defined with their specific rules and transactions, we initially define only one, with *workchain_id* $= 0$. This workchain, called Workchain Zero or the basic workchain, is the one used to work with *TON smart contracts* and transfer *TON coins*, also known as *Grams* (cf. Appendix A). Most applications are likely to require only Workchain Zero. Shardchains of the basic workchain will be called *basic shardchains*.

2.1.12. Block generation intervals. We expect a new block to be generated in each shardchain and the masterchain approximately once every five seconds. This will lead to reasonably small transaction confirmation times. New blocks of all shardchains are generated approximately simultaneously; a new block of the masterchain is generated approximately one second later, because it must contain the hashes of the latest blocks of all shardchains.

2.1.13. Using the masterchain to make workchains and shardchains tightly coupled. Once the hash of a block of a shardchain is incorporated into a block of the masterchain, that shardchain block and all its ancestors are considered "canonical", meaning that they can be referenced from the subsequent blocks of all shardchains as something fixed and immutable. In fact, each new shardchain block contains a hash of the most recent masterchain block, and all shardchain blocks referenced from that masterchain block are considered immutable by the new block.

Essentially, this means that a transaction or a message committed in a shardchain block may be safely used in the very next blocks of the other shardchains, without needing to wait for, say, twenty confirmations (i.e., twenty blocks generated after the original block in the same blockchain) before forwarding a message or taking other actions based on a previous transaction, as is common in most proposed "loosely-coupled" systems (cf. **2.8.14**), such as EOS. This ability to use transactions and messages in other shardchains a mere five seconds after being committed is one of the reasons we believe our "tightly-coupled" system, the first of its kind, will be able to deliver unprecedented performance (cf. **2.8.12** and **2.8.14**).

2.1.14. Masterchain block hash as a global state. According to **2.1.13**, the hash of the last masterchain block completely determines the overall state of the system from the perspective of an external observer. One does not need to monitor the state of all shardchains separately.

2.1.15. Generation of new blocks by validators; cf. 2.6. The TON Blockchain uses a Proof-of-Stake (PoS) approach for generating new blocks in the shardchains and the masterchain. This means that there is a set of, say, up to a few hundred *validators*—special nodes that have deposited *stakes* (large amounts of TON coins) by a special masterchain transaction to be eligible for new block generation and validation.

Then a smaller subset of validators is assigned to each shard (w, s) in a deterministic pseudorandom way, changing approximately every 1024 blocks. This subset of validators suggests and reaches consensus on what the next shardchain block would be, by collecting suitable proposed transactions from the clients into new valid block candidates. For each block, there is a pseudorandomly chosen order on the validators to determine whose block candidate has the highest priority to be committed at each turn.

Validators and other nodes check the validity of the proposed block candidates; if a validator signs an invalid block candidate, it may be automatically punished by losing part or all of its stake, or by being suspended from the set of validators for some time. After that, the validators should reach consensus on the choice of the next block, essentially by an efficient variant of the BFT (Byzantine Fault Tolerant; cf. **2.8.4**) consensus protocol, similar to PBFT [4] or Honey Badger BFT [11]. If consensus is reached, a new block is created, and validators divide between themselves the transaction fees for the transactions included, plus some newly-created ("minted") coins.

Each validator can be elected to participate in several validator subsets; in this case, it is expected to run all validation and consensus algorithms in parallel.

After all new shardchain blocks are generated or a timeout is passed, a new masterchain block is generated, including the hashes of the latest blocks of all shardchains. This is done by BFT consensus of *all* validators.[2]

More detail on the TON PoS approach and its economical model is provided in section **2.6**.

[2]Actually, two-thirds by stake is enough to achieve consensus, but an effort is made to collect as many signatures as possible.

2.1.16. Forks of the masterchain. A complication that arises from our tightly-coupled approach is that switching to a different fork in the masterchain will almost necessarily require switching to another fork in at least some of the shardchains. On the other hand, as long as there are no forks in the masterchain, no forks in the shardchain are even possible, because no blocks in the alternative forks of the shardchains can become "canonical" by having their hashes incorporated into a masterchain block.

The general rule is that *if masterchain block B' is a predecessor of B, B' includes hash $\textsc{Hash}(B'_{w,s})$ of (w,s)-shardchain block $B'_{w,s}$, and B includes hash $\textsc{Hash}(B_{w,s})$, then $B'_{w,s}$* **must** *be a predecessor of $B_{w,s}$; otherwise, the masterchain block B is invalid.*

We expect masterchain forks to be rare, next to non-existent, because in the BFT paradigm adopted by the TON Blockchain they can happen only in the case of incorrect behavior by a *majority* of validators (cf. **2.6.1** and **2.6.15**), which would imply significant stake losses by the offenders. Therefore, no true forks in the shardchains should be expected. Instead, if an invalid shardchain block is detected, it will be corrected by means of the "vertical blockchain" mechanism of the 2-blockchain (cf. **2.1.17**), which can achieve this goal without forking the "horizontal blockchain" (i.e., the shardchain). The same mechanism can be used to fix non-fatal mistakes in the masterchain blocks as well.

2.1.17. Correcting invalid shardchain blocks. Normally, only valid shardchain blocks will be committed, because validators assigned to the shardchain must reach a two-thirds Byzantine consensus before a new block can be committed. However, the system must allow for detection of previously committed invalid blocks and their correction.

Of course, once an invalid shardchain block is found—either by a validator (not necessarily assigned to this shardchain) or by a "fisherman" (any node of the system that made a certain deposit to be able to raise questions about block validity; cf. **2.6.4**)—the invalidity claim and its proof are committed into the masterchain, and the validators that have signed the invalid block are punished by losing part of their stake and/or being temporarily suspended from the set of validators (the latter measure is important for the case of an attacker stealing the private signing keys of an otherwise benign validator).

However, this is not sufficient, because the overall state of the system (TON Blockchain) turns out to be invalid because of the invalid shardchain block previously committed. This invalid block must be replaced by a newer

valid version.

Most systems would achieve this by "rolling back" to the last block before the invalid one in this shardchain and the last blocks unaffected by messages propagated from the invalid block in each of the other shardchains, and creating a new fork from these blocks. This approach has the disadvantage that a large number of otherwise correct and committed transactions are suddenly rolled back, and it is unclear whether they will be included later at all.

The TON Blockchain solves this problem by making each "block" of each shardchain and of the masterchain ("horizontal blockchains") a small blockchain ("vertical blockchain") by itself, containing different versions of this "block", or their "differences". Normally, the vertical blockchain consists of exactly one block, and the shardchain looks like a classical blockchain. However, once the invalidity of a block is confirmed and committed into a masterchain block, the "vertical blockchain" of the invalid block is allowed to grow by a new block in the vertical direction, replacing or editing the invalid block. The new block is generated by the current validator subset for the shardchain in question.

The rules for a new "vertical" block to be valid are quite strict. In particular, if a virtual "account-chain block" (cf. **2.1.2**) contained in the invalid block is valid by itself, it must be left unchanged by the new vertical block.

Once a new "vertical" block is committed on top of the invalid block, its hash is published in a new masterchain block (or rather in a new "vertical" block, lying above the original masterchain block where the hash of the invalid shardchain block was originally published), and the changes are propagated further to any shardchain blocks referring to the previous version of this block (e.g., those having received messages from the incorrect block). This is fixed by committing new "vertical" blocks in vertical blockchains for all blocks previously referring to the "incorrect" block; new vertical blocks will refer to the most recent (corrected) versions instead. Again, strict rules forbid changing account-chains that are not really affected (i.e., that receive the same messages as in the previous version). In this way, fixing an incorrect block generates "ripples" that are ultimately propagated towards the most recent blocks of all affected shardchains; these changes are reflected in new "vertical" masterchain blocks as well.

Once the "history rewriting" ripples reach the most recent blocks, the new shardchain blocks are generated in one version only, being successors of the newest block versions only. This means that they will contain references

to the correct (most recent) vertical blocks from the very beginning.

The masterchain state implicitly defines a map transforming the hash of the first block of each "vertical" blockchain into the hash of its latest version. This enables a client to identify and locate any vertical blockchain by the hash of its very first (and usually the only) block.

2.1.18. TON coins and multi-currency workchains. The TON Blockchain supports up to 2^{32} different "cryptocurrencies", "coins", or "tokens", distinguished by a 32-bit *currency_id*. New cryptocurrencies can be added by special transactions in the masterchain. Each workchain has a basic cryptocurrency, and can have several additional cryptocurrencies.

There is one special cryptocurrency with *currency_id* $= 0$, namely, the *TON coin*, also known as the *Gram* (cf. Appendix A). It is the basic cryptocurrency of Workchain Zero. It is also used for transaction fees and validator stakes.

In principle, other workchains may collect transaction fees in other tokens. In this case, some smart contract for automated conversion of these transaction fees into Grams should be provided.

2.1.19. Messaging and value transfer. Shardchains belonging to the same or different workchains may send *messages* to each other. While the exact form of the messages allowed depends on the receiving workchain and receiving account (smart contract), there are some common fields making inter-workchain messaging possible. In particular, each message may have some *value* attached, in the form of a certain amount of Grams (TON coins) and/or other registered cryptocurrencies, provided they are declared as acceptable cryptocurrencies by the receiving workchain.

The simplest form of such messaging is a value transfer from one (usually not a smart-contract) account to another.

2.1.20. TON Virtual Machine. The *TON Virtual Machine*, also abbreviated as *TON VM* or *TVM*, is the virtual machine used to execute smart-contract code in the masterchain and in the basic workchain. Other workchains may use other virtual machines alongside or instead of the TVM.

Here we list some of its features. They are discussed further in **2.3.12**, **2.3.14** and elsewhere.

- TVM represents all data as a collection of *(TVM) cells* (cf. **2.3.14**). Each cell contains up to 128 data bytes and up to 4 references to other

cells. As a consequence of the "everything is a bag of cells" philosophy (cf. **2.5.14**), this enables TVM to work with all data related to the TON Blockchain, including blocks and blockchain global state if necessary.

- TVM can work with values of arbitrary algebraic data types (cf. **2.3.12**), represented as trees or directed acyclic graphs of TVM cells. However, it is agnostic towards the existence of algebraic data types; it just works with cells.
- TVM has built-in support for hashmaps (cf. **2.3.7**).
- TVM is a stack machine. Its stack keeps either 64-bit integers or cell references.
- 64-bit, 128-bit and 256-bit arithmetic is supported. All n-bit arithmetic operations come in three flavors: for unsigned integers, for signed integers and for integers modulo 2^n (no automatic overflow checks in the latter case).
- TVM has unsigned and signed integer conversion from n-bit to m-bit, for all $0 \leq m, n \leq 256$, with overflow checks.
- All arithmetic operations perform overflow checks by default, greatly simplifying the development of smart contracts.
- TVM has "multiply-then-shift" and "shift-then-divide" arithmetic operations with intermediate values computed in a larger integer type; this simplifies implementing fixed-point arithmetic.
- TVM offers support for bit strings and byte strings.
- Support for 256-bit Elliptic Curve Cryptography (ECC) for some predefined curves, including Curve25519, is present.
- Support for Weil pairings on some elliptic curves, useful for fast implementation of zk-SNARKs, is also present.
- Support for popular hash functions, including SHA256, is present.
- TVM can work with Merkle proofs (cf. **5.1.9**).

- TVM offers support for "large" or "global" smart contracts. Such smart contracts must be aware of sharding (cf. **2.3.18** and **2.3.16**). Usual (local) smart contracts can be sharding-agnostic.
- TVM supports closures.
- A "spineless tagless G-machine" [13] can be easily implemented inside TVM.

Several high-level languages can be designed for TVM, in addition to the "TVM assembly". All these languages will have static types and will support algebraic data types. We envision the following possibilities:

- A Java-like imperative language, with each smart contract resembling a separate class.
- A lazy functional language (think of Haskell).
- An eager functional language (think of ML).

2.1.21. Configurable parameters. An important feature of the TON Blockchain is that many of its parameters are *configurable*. This means that they are part of the masterchain state, and can be changed by certain special proposal/vote/result transactions in the masterchain, without any need for hard forks. Changing such parameters will require collecting two-thirds of validator votes and more than half of the votes of all other participants who would care to take part in the voting process in favor of the proposal.

2.2 Generalities on Blockchains

2.2.1. General blockchain definition. In general, any *(true) blockchain* is a sequence of *blocks*, each block B containing a reference $\textsc{blk-prev}(B)$ to the previous block (usually by including the hash of the previous block into the header of the current block), and a list of *transactions*. Each transaction describes some transformation of the *global blockchain state*; the transactions listed in a block are applied sequentially to compute the new state starting from the old state, which is the resulting state after the evaluation of the previous block.

2.2.2. Relevance for the TON Blockchain. Recall that the *TON Blockchain* is not a true blockchain, but a collection of 2-blockchains (i.e., of blockchains of blockchains; cf. **2.1.1**), so the above is not directly applicable to it. However, we start with these generalities on true blockchains to use them as building blocks for our more sophisticated constructions.

2.2.3. Blockchain instance and blockchain type. One often uses the word *blockchain* to denote both a general *blockchain type* and its specific *blockchain instances*, defined as sequences of blocks satisfying certain conditions. For example, **2.2.1** refers to blockchain instances.

In this way, a blockchain type is usually a "subtype" of the type $Block^*$ of lists (i.e., finite sequences) of blocks, consisting of those sequences of blocks that satisfy certain compatibility and validity conditions:

$$Blockchain \subset Block^* \tag{1}$$

A better way to define *Blockchain* would be to say that *Blockchain* is a *dependent couple type*, consisting of couples $(\mathbb{B}, v)$, with first component $\mathbb{B} : Block^*$ being of type $Block^*$ (i.e., a list of blocks), and the second component $v : isValidBc(\mathbb{B})$ being a proof or a witness of the validity of $\mathbb{B}$. In this way,

$$Blockchain \equiv \Sigma_{(\mathbb{B}:Block^*)} isValidBc(\mathbb{B}) \tag{2}$$

We use here the notation for dependent sums of types borrowed from [16].

2.2.4. Dependent type theory, Coq and TL. Note that we are using (Martin-Löf) dependent type theory here, similar to that used in the Coq[3] proof assistant. A simplified version of dependent type theory is also used in *TL (Type Language)*,[4] which will be used in the formal specification of the TON Blockchain to describe the serialization of all data structures and the layouts of blocks, transactions, and the like.

In fact, dependent type theory gives a useful formalization of what a proof is, and such formal proofs (or their serializations) might become handy when one needs to provide proof of invalidity for some block, for example.

2.2.5. TL, or the Type Language. Since TL (Type Language) will be used in the formal specifications of TON blocks, transactions, and network datagrams, it warrants a brief discussion.

[3] `https://coq.inria.fr`

[4] `https://core.telegram.org/mtproto/TL`

TL is a language suitable for description of dependent algebraic *types*, which are allowed to have numeric (natural) and type parameters. Each type is described by means of several *constructors*. Each constructor has a (human-readable) identifier and a *name,* which is a bit string (32-bit integer by default). Apart from that, the definition of a constructor contains a list of fields along with their types.

A collection of constructor and type definitions is called a *TL-scheme*. It is usually kept in one or several files with the suffix `.tl`.

An important feature of TL-schemes is that they determine an unambiguous way of serializing and deserializing values (or objects) of algebraic types defined. Namely, when a value needs to be serialized into a stream of bytes, first the name of the constructor used for this value is serialized. Recursively computed serializations of each field follow.

The description of a previous version of TL, suitable for serializing arbitrary objects into sequences of 32-bit integers, is available at `https://core.telegram.org/mtproto/TL`. A new version of TL, called *TL-B*, is being developed for the purpose of describing the serialization of objects used by the TON Project. This new version can serialize objects into streams of bytes and even bits (not just 32-bit integers), and offers support for serialization into a tree of TVM cells (cf. **2.3.14**). A description of TL-B will be a part of the formal specification of the TON Blockchain.

2.2.6. Blocks and transactions as state transformation operators. Normally, any blockchain (type) *Blockchain* has an associated global state (type) *State*, and a transaction (type) *Transaction*. The semantics of a blockchain are to a large extent determined by the transaction application function:

$$ev\_trans' : Transaction \times State \to State^{?} \tag{3}$$

Here $X^{?}$ denotes $\textsc{Maybe}\, X$, the result of applying the Maybe monad to type X. This is similar to our use of X^* for $\textsc{List}\, X$. Essentially, a value of type $X^{?}$ is either a value of type X or a special value $\perp$ indicating the absence of an actual value (think about a null pointer). In our case, we use $State^{?}$ instead of *State* as the result type because a transaction may be invalid if invoked from certain original states (think about attempting to withdraw from an account more money than it is actually there).

We might prefer a curried version of ev_trans':

$$ev\_trans : Transaction \to State \to State^{?} \tag{4}$$

Because a block is essentially a list of transactions, the block evaluation function

$$ev\_block : Block \to State \to State^{?} \tag{5}$$

can be derived from *ev_trans*. It takes a block $B : Block$ and the previous blockchain state $s : State$ (which might include the hash of the previous block) and computes the next blockchain state $s' = ev\_block(B)(s) : State$, which is either a true state or a special value $\perp$ indicating that the next state cannot be computed (i.e., that the block is invalid if evaluated from the starting state given—for example, the block includes a transaction trying to debit an empty account.)

2.2.7. Block sequence numbers. Each block B in the blockchain can be referred to by its *sequence number* $\text{BLK-SEQNO}(B)$, starting from zero for the very first block, and incremented by one whenever passing to the next block. More formally,

$$\text{BLK-SEQNO}(B) = \text{BLK-SEQNO}\big(\text{BLK-PREV}(B)\big) + 1 \tag{6}$$

Notice that the sequence number does not identify a block uniquely in the presence of *forks*.

2.2.8. Block hashes. Another way of referring to a block B is by its hash $\text{BLK-HASH}(B)$, which is actually the hash of the *header* of block B (however, the header of the block usually contains hashes that depend on all content of block B). Assuming that there are no collisions for the hash function used (or at least that they are very improbable), a block is uniquely identified by its hash.

2.2.9. Hash assumption. During formal analysis of blockchain algorithms, we assume that there are no collisions for the k-bit hash function $\text{HASH} : Bytes^* \to \mathbf{2}^k$ used:

$$\text{HASH}(s) = \text{HASH}(s') \Rightarrow s = s' \quad \text{for any } s,\, s' \in Bytes^* \tag{7}$$

Here $Bytes = \{0 \ldots 255\} = \mathbf{2}^8$ is the type of bytes, or the set of all byte values, and $Bytes^*$ is the type or set of arbitrary (finite) lists of bytes; while $\mathbf{2} = \{0, 1\}$ is the bit type, and $\mathbf{2}^k$ is the set (or actually the type) of all k-bit sequences (i.e., of k-bit numbers).

Of course, (7) is impossible mathematically, because a map from an infinite set to a finite set cannot be injective. A more rigorous assumption would be

$$\forall s, s' : s \neq s', P\big(\textsc{Hash}(s) = \textsc{Hash}(s')\big) = 2^{-k} \tag{8}$$

However, this is not so convenient for the proofs. If (8) is used at most N times in a proof with $2^{-k}N < \epsilon$ for some small ϵ (say, $\epsilon = 10^{-18}$), we can reason as if (7) were true, provided we accept a failure probability ϵ (i.e., the final conclusions will be true with probability at least $1 - \epsilon$).

Final remark: in order to make the probability statement of (8) really rigorous, one must introduce a probability distribution on the set $Bytes^*$ of all byte sequences. A way of doing this is by assuming all byte sequences of the same length l equiprobable, and setting the probability of observing a sequence of length l equal to $p^l - p^{l+1}$ for some $p \to 1-$. Then (8) should be understood as a limit of conditional probability $P\big(\textsc{Hash}(s) = \textsc{Hash}(s') | s \neq s'\big)$ when p tends to one from below.

2.2.10. Hash used for the TON Blockchain. We are using the 256-bit SHA256 hash for the TON Blockchain for the time being. If it turns out to be weaker than expected, it can be replaced by another hash function in the future. The choice of the hash function is a configurable parameter of the protocol, so it can be changed without hard forks as explained in **2.1.21**.

2.3 Blockchain State, Accounts and Hashmaps

We have noted above that any blockchain defines a certain global state, and each block and each transaction defines a transformation of this global state. Here we describe the global state used by TON blockchains.

2.3.1. Account IDs. The basic account IDs used by TON blockchains—or at least by its masterchain and Workchain Zero—are 256-bit integers, assumed to be public keys for 256-bit Elliptic Curve Cryptography (ECC) for a specific elliptic curve. In this way,

$$account\_id : Account = uint_{256} = \mathbf{2}^{256} \tag{9}$$

Here *Account* is the account *type*, while $account\_id : Account$ is a specific variable of type *Account*.

Other workchains can use other account ID formats, 256-bit or otherwise. For example, one can use Bitcoin-style account IDs, equal to SHA256 of an ECC public key.

However, the bit length l of an account ID must be fixed during the creation of the workchain (in the masterchain), and it must be at least 64, because the first 64 bits of *account_id* are used for sharding and message routing.

2.3.2. Main component: *Hashmaps*. The principal component of the TON blockchain state is a *hashmap*. In some cases we consider (partially defined) "maps" $h : \mathbf{2}^n \dashrightarrow \mathbf{2}^m$. More generally, we might be interested in hashmaps $h : \mathbf{2}^n \dashrightarrow X$ for a composite type X. However, the source (or index) type is almost always $\mathbf{2}^n$.

Sometimes, we have a "default value" $empty : X$, and the hashmap $h : \mathbf{2}^n \to X$ is "initialized" by its "default value" $i \mapsto empty$.

2.3.3. Example: TON account balances. An important example is given by TON account balances. It is a hashmap

$$balance : Account \to uint_{128} \tag{10}$$

mapping $Account = \mathbf{2}^{256}$ into a Gram (TON coin) balance of type $uint_{128} = \mathbf{2}^{128}$. This hashmap has a default value of zero, meaning that initially (before the first block is processed) the balance of all accounts is zero.

2.3.4. Example: smart-contract persistent storage. Another example is given by smart-contract persistent storage, which can be (very approximately) represented as a hashmap

$$storage : \mathbf{2}^{256} \dashrightarrow \mathbf{2}^{256} \tag{11}$$

This hashmap also has a default value of zero, meaning that uninitialized cells of persistent storage are assumed to be zero.

2.3.5. Example: persistent storage of all smart contracts. Because we have more than one smart contract, distinguished by *account_id*, each having its separate persistent storage, we must actually have a hashmap

$$Storage : Account \dashrightarrow (\mathbf{2}^{256} \dashrightarrow \mathbf{2}^{256}) \tag{12}$$

mapping *account_id* of a smart contract into its persistent storage.

2.3.6. Hashmap type. The hashmap is not just an abstract (partially defined) function $\mathbf{2}^n \dashrightarrow X$; it has a specific representation. Therefore, we suppose that we have a special hashmap type

$$\mathit{Hashmap}(n, X) : \mathit{Type} \tag{13}$$

corresponding to a data structure encoding a (partial) map $\mathbf{2}^n \dashrightarrow X$. We can also write

$$\mathit{Hashmap}(n : \mathit{nat})(X : \mathit{Type}) : \mathit{Type} \tag{14}$$

or

$$\mathit{Hashmap} : \mathit{nat} \to \mathit{Type} \to \mathit{Type} \tag{15}$$

We can always transform $h : \mathit{Hashmap}(n, X)$ into a map $\mathit{hget}(h) : \mathbf{2}^n \to X^?$. Henceforth, we usually write $h[i]$ instead of $\mathit{hget}(h)(i)$:

$$h[i] :\equiv \mathit{hget}(h)(i) : X^? \quad \text{for any } i : \mathbf{2}^n,\ h : \mathit{Hashmap}(n, X) \tag{16}$$

2.3.7. Definition of hashmap type as a Patricia tree. Logically, one might define $\mathit{Hashmap}(n, X)$ as an (incomplete) binary tree of depth n with edge labels 0 and 1 and with values of type X in the leaves. Another way to describe the same structure would be as a *(bitwise) trie* for binary strings of length equal to n.

In practice, we prefer to use a compact representation of this trie, by compressing each vertex having only one child with its parent. The resulting representation is known as a *Patricia tree* or a *binary radix tree*. Each intermediate vertex now has exactly two children, labeled by two non-empty binary strings, beginning with zero for the left child and with one for the right child.

In other words, there are two types of (non-root) nodes in a Patricia tree:

- $\textsc{Leaf}(x)$, containing value x of type X.
- $\textsc{Node}(l, s_l, r, s_r)$, where l is the (reference to the) left child or subtree, s_l is the bitstring labeling the edge connecting this vertex to its left child (always beginning with 0), r is the right subtree, and s_r is the bitstring labeling the edge to the right child (always beginning with 1).

A third type of node, to be used only once at the root of the Patricia tree, is also necessary:

- $\textsc{Root}(n, s_0, t)$, where n is the common length of index bitstrings of $\mathit{Hashmap}(n, X)$, s_0 is the common prefix of all index bitstrings, and t is a reference to a Leaf or a Node.

If we want to allow the Patricia tree to be empty, a fourth type of (root) node would be used:

- $\textsc{EmptyRoot}(n)$, where n is the common length of all index bitstrings.

We define the height of a Patricia tree by

$$\textsc{height}(\textsc{Leaf}(x)) = 0 \tag{17}$$

$$\textsc{height}\big(\textsc{Node}(l, s_l, r, s_r)\big) = \textsc{height}(l) + \textsc{len}(s_l) = \textsc{height}(r) + \textsc{len}(s_r) \tag{18}$$

$$\textsc{height}\big(\textsc{Root}(n, s_0, t)\big) = \textsc{len}(s_0) + \textsc{height}(t) = n \tag{19}$$

The last two expressions in each of the last two formulas must be equal. We use Patricia trees of height n to represent values of type $\mathit{Hashmap}(n, X)$.

If there are N leaves in the tree (i.e., our hashmap contains N values), then there are exactly $N - 1$ intermediate vertices. Inserting a new value always involves splitting an existing edge by inserting a new vertex in the middle and adding a new leaf as the other child of this new vertex. Deleting a value from a hashmap does the opposite: a leaf and its parent are deleted, and the parent's parent and its other child become directly linked.

2.3.8. Merkle-Patricia trees. When working with blockchains, we want to be able to compare Patricia trees (i.e., hash maps) and their subtrees, by reducing them to a single hash value. The classical way of achieving this is given by the Merkle tree. Essentially, we want to describe a way of hashing objects h of type $\mathit{Hashmap}(n, X)$ with the aid of a hash function Hash defined for binary strings, provided we know how to compute hashes $\textsc{Hash}(x)$ of objects $x : X$ (e.g., by applying the hash function Hash to a binary serialization of object x).

One might define $\textsc{Hash}(h)$ recursively as follows:

$$\textsc{Hash}\big(\textsc{Leaf}(x)\big) := \textsc{Hash}(x) \tag{20}$$

$$\textsc{Hash}\big(\textsc{Node}(l, s_l, r, s_r)\big) := \textsc{Hash}\big(\textsc{Hash}(l).\,\textsc{Hash}(r).\,\textsc{code}(s_l).\,\textsc{code}(s_r)\big) \tag{21}$$

$$\textsc{Hash}\big(\textsc{Root}(n, s_0, t)\big) := \textsc{Hash}\big(\textsc{code}(n).\,\textsc{code}(s_0).\,\textsc{Hash}(t)\big) \tag{22}$$

Here $s.t$ denotes the concatenation of (bit) strings s and t, and $\text{CODE}(s)$ is a prefix code for all bit strings s. For example, one might encode 0 by 10, 1 by 11, and the end of the string by 0.[5]

We will see later (cf. **2.3.12** and **2.3.14**) that this is a (slightly tweaked) version of recursively defined hashes for values of arbitrary (dependent) algebraic types.

2.3.9. Recomputing Merkle tree hashes. This way of recursively defining $\text{HASH}(h)$, called a *Merkle tree hash*, has the advantage that, if one explicitly stores $\text{HASH}(h')$ along with each node h' (resulting in a structure called a *Merkle tree*, or, in our case, a *Merkle–Patricia tree*), one needs to recompute only at most n hashes when an element is added to, deleted from or changed in the hashmap.

In this way, if one represents the global blockchain state by a suitable Merkle tree hash, it is easy to recompute this state hash after each transaction.

2.3.10. Merkle proofs. Under the assumption (7) of "injectivity" of the chosen hash function HASH, one can construct a proof that, for a given value z of $\text{HASH}(h)$, $h : \mathit{Hashmap}(n, X)$, one has $\mathit{hget}(h)(i) = x$ for some $i : \mathbf{2}^n$ and $x : X$. Such a proof will consist of the path in the Merkle–Patricia tree from the leaf corresponding to i to the root, augmented by the hashes of all siblings of all nodes occurring on this path.

In this way, a light node[6] knowing only the value of $\text{HASH}(h)$ for some hashmap h (e.g., smart-contract persistent storage or global blockchain state) might request from a full node[7] not only the value $x = h[i] = \mathit{hget}(h)(i)$, but such a value along with a Merkle proof starting from the already known value

[5] One can show that this encoding is optimal for approximately half of all edge labels of a Patricia tree with random or consecutive indices. Remaining edge labels are likely to be long (i.e., almost 256 bits long). Therefore, a nearly optimal encoding for edge labels is to use the above code with prefix 0 for "short" bit strings, and encode 1, then nine bits containing length $l = |s|$ of bitstring s, and then the l bits of s for "long" bitstrings (with $l \geq 10$).

[6] A *light node* is a node that does not keep track of the full state of a shardchain; instead, it keeps minimal information such as the hashes of the several most recent blocks, and relies on information obtained from full nodes when it becomes necessary to inspect some parts of the full state.

[7] A *full node* is a node keeping track of the complete up-to-date state of the shardchain in question.

Hash(h). Then, under assumption (7), the light node can check for itself that x is indeed the correct value of $h[i]$.

In some cases, the client may want to obtain the value $y = \text{Hash}(x) = \text{Hash}(h[i])$ instead—for example, if x itself is very large (e.g., a hashmap itself). Then a Merkle proof for (i, y) can be provided instead. If x is a hashmap as well, then a second Merkle proof starting from $y = \text{Hash}(x)$ may be obtained from a full node, to provide a value $x[j] = h[i][j]$ or just its hash.

2.3.11. Importance of Merkle proofs for a multi-chain system such as TON. Notice that a node normally cannot be a full node for all shardchains existing in the TON environment. It usually is a full node only for some shardchains—for instance, those containing its own account, a smart contract it is interested in, or those that this node has been assigned to be a validator of. For other shardchains, it must be a light node—otherwise the storage, computing and network bandwidth requirements would be prohibitive. This means that such a node cannot directly check assertions about the state of other shardchains; it must rely on Merkle proofs obtained from full nodes for those shardchains, which is as safe as checking by itself unless (7) fails (i.e., a hash collision is found).

2.3.12. Peculiarities of TON VM. The TON VM or TVM (Telegram Virtual Machine), used to run smart contracts in the masterchain and Workchain Zero, is considerably different from customary designs inspired by the EVM (Ethereum Virtual Machine): it works not just with 256-bit integers, but actually with (almost) arbitrary "records", "structures", or "sum-product types", making it more suitable to execute code written in high-level (especially functional) languages. Essentially, TVM uses tagged data types, not unlike those used in implementations of Prolog or Erlang.

One might imagine first that the state of a TVM smart contract is not just a hashmap $\mathbf{2}^{256} \to \mathbf{2}^{256}$, or $\mathit{Hashmap}(256, \mathbf{2}^{256})$, but (as a first step) $\mathit{Hashmap}(256, X)$, where X is a type with several constructors, enabling it to store, apart from 256-bit integers, other data structures, including other hashmaps $\mathit{Hashmap}(256, X)$ in particular. This would mean that a cell of TVM (persistent or temporary) storage—or a variable or an element of an array in a TVM smart-contract code—may contain not only an integer, but a whole new hashmap. Of course, this would mean that a cell contains not just 256 bits, but also, say, an 8-bit tag, describing how these 256 bits should be interpreted.

In fact, values do not need to be precisely 256-bit. The value format used by TVM consists of a sequence of raw bytes and references to other structures, mixed in arbitrary order, with some descriptor bytes inserted in suitable locations to be able to distinguish pointers from raw data (e.g., strings or integers); cf. **2.3.14**.

This raw value format may be used to implement arbitrary sum-product algebraic types. In this case, the value would contain a raw byte first, describing the "constructor" being used (from the perspective of a high-level language), and then other "fields" or "constructor arguments", consisting of raw bytes and references to other structures depending on the constructor chosen (cf. **2.2.5**). However, TVM does not know anything about the correspondence between constructors and their arguments; the mixture of bytes and references is explicitly described by certain descriptor bytes.[8]

The Merkle tree hashing is extended to arbitrary such structures: to compute the hash of such a structure, all references are recursively replaced by hashes of objects referred to, and then the hash of the resulting byte string (descriptor bytes included) is computed.

In this way, the Merkle tree hashing for hashmaps, described in **2.3.8**, is just a special case of hashing for arbitrary (dependent) algebraic data types, applied to type $\mathit{Hashmap}(n, X)$ with two constructors.[9]

2.3.13. Persistent storage of TON smart contracts. Persistent storage of a TON smart contract essentially consists of its "global variables", preserved between calls to the smart contract. As such, it is just a "product", "tuple", or "record" type, consisting of fields of the correct types, corresponding to one global variable each. If there are too many global variables, they cannot fit into one TON cell because of the global restriction on TON cell size. In such a case, they are split into several records and organized into a tree, essentially becoming a "product of products" or "product of products of products" type instead of just a product type.

2.3.14. TVM Cells. Ultimately, the TON VM keeps all data in a collection of *(TVM) cells*. Each cell contains two descriptor bytes first, indicating how

[8]These two descriptor bytes, present in any TVM cell, describe only the total number of references and the total number of raw bytes; references are kept together either before or after all raw bytes.

[9]Actually, Leaf and Node are constructors of an auxiliary type, $\mathit{HashmapAux}(n, X)$. Type $\mathit{Hashmap}(n, X)$ has constructors Root and EmptyRoot, with Root containing a value of type $\mathit{HashmapAux}(n, X)$.

many bytes of raw data are present in this cell (up to 128) and how many references to other cells are present (up to four). Then these raw data bytes and references follow. Each cell is referenced exactly once, so we might have included in each cell a reference to its "parent" (the only cell referencing this one). However, this reference need not be explicit.

In this way, the persistent data storage cells of a TON smart contract are organized into a tree,[10] with a reference to the root of this tree kept in the smart-contract description. If necessary, a Merkle tree hash of this entire persistent storage is recursively computed, starting from the leaves and then simply replacing all references in a cell with the recursively computed hashes of the referenced cells, and subsequently computing the hash of the byte string thus obtained.

2.3.15. Generalized Merkle proofs for values of arbitrary algebraic types. Because the TON VM represents a value of arbitrary algebraic type by means of a tree consisting of (TVM) cells, and each cell has a well-defined (recursively computed) Merkle hash, depending in fact on the whole subtree rooted in this cell, we can provide "generalized Merkle proofs" for (parts of) values of arbitrary algebraic types, intended to prove that a certain subtree of a tree with a known Merkle hash takes a specific value or a value with a specific hash. This generalizes the approach of **2.3.10**, where only Merkle proofs for $x[i] = y$ have been considered.

2.3.16. Support for sharding in TON VM data structures. We have just outlined how the TON VM, without being overly complicated, supports arbitrary (dependent) algebraic data types in high-level smart-contract languages. However, sharding of large (or global) smart contracts requires special support on the level of TON VM. To this end, a special version of the hashmap type has been added to the system, amounting to a "map" $\mathit{Account} \dashrightarrow X$. This "map" might seem equivalent to $\mathit{Hashmap}(m, X)$, where $\mathit{Account} = \mathbf{2}^m$. However, when a shard is split in two, or two shards are merged, such hashmaps are automatically split in two, or merged back, so as to keep only those keys that belong to the corresponding shard.

2.3.17. Payment for persistent storage. A noteworthy feature of the TON Blockchain is the payment exacted from smart contracts for storing

[10]Logically; the "bag of cells" representation described in **2.5.5** identifies all duplicate cells, transforming this tree into a directed acyclic graph (dag) when serialized.

their persistent data (i.e., for enlarging the total state of the blockchain). It works as follows:

Each block declares two rates, nominated in the principal currency of the blockchain (usually the Gram): the price for keeping one cell in the persistent storage, and the price for keeping one raw byte in some cell of the persistent storage. Statistics on the total numbers of cells and bytes used by each account are stored as part of its state, so by multiplying these numbers by the two rates declared in the block header, we can compute the payment to be deducted from the account balance for keeping its data between the previous block and the current one.

However, payment for persistent storage usage is not exacted for every account and smart contract in each block; instead, the sequence number of the block where this payment was last exacted is stored in the account data, and when any action is done with the account (e.g., a value transfer or a message is received and processed by a smart contract), the storage usage payment for all blocks since the previous such payment is deducted from the account balance before performing any further actions. If the account's balance would become negative after this, the account is destroyed.

A workchain may declare some number of raw data bytes per account to be "free" (i.e., not participating in the persistent storage payments) in order to make "simple" accounts, which keep only their balance in one or two cryptocurrencies, exempt from these constant payments.

Notice that, if nobody sends any messages to an account, its persistent storage payments are not collected, and it can exist indefinitely. However, anybody can send, for instance, an empty message to destroy such an account. A small incentive, collected from part of the original balance of the account to be destroyed, can be given to the sender of such a message. We expect, however, that the validators would destroy such insolvent accounts for free, simply to decrease the global blockchain state size and to avoid keeping large amounts of data without compensation.

Payments collected for keeping persistent data are distributed among the validators of the shardchain or the masterchain (proportionally to their stakes in the latter case).

2.3.18. Local and global smart contracts; smart-contract instances. A smart contract normally resides just in one shard, selected according to the smart contract's *account_id*, similarly to an "ordinary" account. This is usually sufficient for most applications. However, some "high-load" smart con-

tracts may want to have an "instance" in each shardchain of some workchain. To achieve this, they must propagate their creating transaction into all shardchains, for instance, by committing this transaction into the "root" shardchain $(w, \emptyset)$[11] of the workchain w and paying a large commission.[12]

This action effectively creates instances of the smart contract in each shard, with separate balances. Originally, the balance transferred in the creating transaction is distributed simply by giving the instance in shard (w, s) the $2^{-|s|}$ part of the total balance. When a shard splits into two child shards, balances of all instances of global smart contracts are split in half; when two shards merge, balances are added together.

In some cases, splitting/merging instances of global smart contracts may involve (delayed) execution of special methods of these smart contracts. By default, the balances are split and merged as described above, and some special "account-indexed" hashmaps are also automatically split and merged (cf. **2.3.16**).

2.3.19. Limiting splitting of smart contracts. A global smart contract may limit its splitting depth d upon its creation, in order to make persistent storage expenses more predictable. This means that, if shardchain (w, s) with $|s| \geq d$ splits in two, only one of two new shardchains inherits an instance of the smart contract. This shardchain is chosen deterministically: each global smart contract has some "*account_id*", which is essentially the hash of its creating transaction, and its instances have the same *account_id* with the first $\leq d$ bits replaced by suitable values needed to fall into the correct shard. This *account_id* selects which shard will inherit the smart-contract instance after splitting.

2.3.20. Account/Smart-contract state. We can summarize all of the above to conclude that an account or smart-contract state consists of the following:

- A balance in the principal currency of the blockchain
- A balance in other currencies of the blockchain
- Smart-contract code (or its hash)

[11] A more expensive alternative is to publish such a "global" smart contract in the masterchain.

[12] This is a sort of "broadcast" feature for all shards, and as such, it must be quite expensive.

- Smart-contract persistent data (or its Merkle hash)
- Statistics on the number of persistent storage cells and raw bytes used
- The last time (actually, the masterchain block number) when payment for smart-contract persistent storage was collected
- The public key needed to transfer currency and send messages from this account (optional; by default equal to *account_id* itself). In some cases, more sophisticated signature checking code may be located here, similar to what is done for Bitcoin transaction outputs; then the *account_id* will be equal to the hash of this code.

We also need to keep somewhere, either in the account state or in some other account-indexed hashmap, the following data:

- The output message queue of the account (cf. **2.4.17**)
- The collection of (hashes of) recently delivered messages (cf. **2.4.23**)

Not all of these are really required for every account; for example, smart-contract code is needed only for smart contracts, but not for "simple" accounts. Furthermore, while any account must have a non-zero balance in the principal currency (e.g., Grams for the masterchain and shardchains of the basic workchain), it may have balances of zero in other currencies. In order to avoid keeping unused data, a sum-product type (depending on the workchain) is defined (during the workchain's creation), which uses different tag bytes (e.g., TL constructors; cf. **2.2.5**) to distinguish between different "constructors" used. Ultimately, the account state is itself kept as a collection of cells of the TVM persistent storage.

2.4 Messages Between Shardchains

An important component of the TON Blockchain is the *messaging system* between blockchains. These blockchains may be shardchains of the same workchain, or of different workchains.

2.4.1. Messages, accounts and transactions: a bird's eye view of the system. *Messages* are sent from one account to another. Each *transaction* consists of an account receiving one message, changing its state according to certain rules, and generating several (maybe one or zero) new messages to

other accounts. Each message is generated and received (delivered) exactly once.

This means that messages play a fundamental role in the system, comparable to that of accounts (smart contracts). From the perspective of the Infinite Sharding Paradigm (cf. **2.1.2**), each account resides in its separate "account-chain", and the only way it can affect the state of some other account is by sending a message.

2.4.2. Accounts as processes or actors; Actor model. One might think about accounts (and smart contracts) as "processes", or "actors", that are able to process incoming messages, change their internal state and generate some outbound messages as a result. This is closely related to the so-called *Actor model*, used in languages such as Erlang (however, actors in Erlang are usually called "processes"). Since new actors (i.e., smart contracts) are also allowed to be created by existing actors as a result of processing an inbound message, the correspondence with the Actor model is essentially complete.

2.4.3. Message recipient. Any message has its *recipient*, characterized by the *target workchain identifier* w (assumed by default to be the same as that of the originating shardchain), and the *recipient account* $account\_id$. The exact format (i.e., number of bits) of $account\_id$ depends on w; however, the shard is always determined by its first (most significant) 64 bits.

2.4.4. Message sender. In most cases, a message has a *sender*, characterized again by a $(w', account\_id')$ pair. If present, it is located after the message recipient and message value. Sometimes, the sender is unimportant or it is somebody outside the blockchain (i.e., not a smart contract), in which case this field is absent.

Notice that the Actor model does not require the messages to have an implicit sender. Instead, messages may contain a reference to the Actor to which an answer to the request should be sent; usually it coincides with the sender. However, it is useful to have an explicit unforgeable sender field in a message in a cryptocurrency (Byzantine) environment.

2.4.5. Message value. Another important characteristic of a message is its attached *value*, in one or several cryptocurrencies supported both by the source and by the target workchain. The value of the message is indicated at its very beginning immediately after the message recipient; it is essentially a list of $(currency\_id, value)$ pairs.

Notice that "simple" value transfers between "simple" accounts are just empty (no-op) messages with some value attached to them. On the other hand, a slightly more complicated message body might contain a simple text or binary comment (e.g., about the purpose of the payment).

2.4.6. External messages, or "messages from nowhere". Some messages arrive into the system "from nowhere"—that is, they are not generated by an account (smart contract or not) residing in the blockchain. The most typical example arises when a user wants to transfer some funds from an account controlled by her to some other account. In this case, the user sends a "message from nowhere" to her own account, requesting it to generate a message to the receiving account, carrying the specified value. If this message is correctly signed, her account receives it and generates the required outbound messages.

In fact, one might consider a "simple" account as a special case of a smart contract with predefined code. This smart contract receives only one type of message. Such an inbound message must contain a list of outbound messages to be generated as a result of delivering (processing) the inbound message, along with a signature. The smart contract checks the signature, and, if it is correct, generates the required messages.

Of course, there is a difference between "messages from nowhere" and normal messages, because the "messages from nowhere" cannot bear value, so they cannot pay for their "gas" (i.e., their processing) themselves. Instead, they are tentatively executed with a small gas limit before even being suggested for inclusion in a new shardchain block; if the execution fails (the signature is incorrect), the "message from nowhere" is deemed incorrect and is discarded. If the execution does not fail within the small gas limit, the message may be included in a new shardchain block and processed completely, with the payment for the gas (processing capacity) consumed exacted from the receiver's account. "Messages from nowhere" can also define some transaction fee which is deducted from the receiver's account on top of the gas payment for redistribution to the validators.

In this sense, "messages from nowhere" or "external messages" take the role of transaction candidates used in other blockchain systems (e.g., Bitcoin and Ethereum).

2.4.7. Log messages, or "messages to nowhere". Similarly, sometimes a special message can be generated and routed to a specific shardchain not

to be delivered to its recipient, but to be logged in order to be easily observable by anybody receiving updates about the shard in question. These logged messages may be output in a user's console, or trigger an execution of some script on an off-chain server. In this sense, they represent the external "output" of the "blockchain supercomputer", just as the "messages from nowhere" represent the external "input" of the "blockchain supercomputer".

2.4.8. Interaction with off-chain services and external blockchains. These external input and output messages can be used for interacting with off-chain services and other (external) blockchains, such as Bitcoin or Ethereum. One might create tokens or cryptocurrencies inside the TON Blockchain pegged to Bitcoins, Ethers or any ERC-20 tokens defined in the Ethereum blockchain, and use "messages from nowhere" and "messages to nowhere", generated and processed by scripts residing on some third-party off-chain servers, to implement the necessary interaction between the TON Blockchain and these external blockchains.

2.4.9. Message body. The *message body* is simply a sequence of bytes, the meaning of which is determined only by the receiving workchain and/or smart contract. For blockchains using TON VM, this could be the serialization of any TVM cell, generated automatically via the `Send()` operation. Such a serialization is obtained simply by recursively replacing all references in a TON VM cell with the cells referred to. Ultimately, a string of raw bytes appears, which is usually prepended by a 4-byte "message type" or "message constructor", used to select the correct method of the receiving smart contract.

Another option would be to use TL-serialized objects (cf. **2.2.5**) as message bodies. This might be especially useful for communication between different workchains, one or both of which are not necessarily using the TON VM.

2.4.10. Gas limit and other workchain/VM-specific parameters. Sometimes a message needs to carry information about the gas limit, the gas price, transaction fees and similar values that depend on the receiving workchain and are relevant only for the receiving workchain, but not necessarily for the originating workchain. Such parameters are included in or before the message body, sometimes (depending on the workchain) with special 4-byte prefixes indicating their presence (which can be defined by a TL-scheme; cf. **2.2.5**).

2.4.11. Creating messages: smart contracts and transactions. There are two sources of new messages. Most messages are created during smart-contract execution (via the `Send()` operation in TON VM), when some smart contract is invoked to process an incoming message. Alternatively, messages may come from the outside as "external messages" or "messages from nowhere" (cf. **2.4.6**).[13]

2.4.12. Delivering messages. When a message reaches the shardchain containing its destination account,[14] it is "delivered" to its destination account. What happens next depends on the workchain; from an outside perspective, it is important that such a message can never be forwarded further from this shardchain.

For shardchains of the basic workchain, delivery consists in adding the message value (minus any gas payments) to the balance of the receiving account, and possibly in invoking a message-dependent method of the receiving smart contract afterwards, if the receiving account is a smart contract. In fact, a smart contract has only one entry point for processing all incoming messages, and it must distinguish between different types of messages by looking at their first few bytes (e.g., the first four bytes containing a TL constructor; cf. **2.2.5**).

2.4.13. Delivery of a message is a transaction. Because the delivery of a message changes the state of an account or smart contract, it is a special *transaction* in the receiving shardchain, and is explicitly registered as such. Essentially, *all* TON Blockchain transactions consist in the delivery of one inbound message to its receiving account (smart contract), neglecting some minor technical details.

2.4.14. Messages between instances of the same smart contract. Recall that a smart contract may be *local* (i.e., residing in one shardchain as any ordinary account does) or *global* (i.e., having instances in all shards, or at least in all shards up to some known depth d; cf. **2.3.18**). Instances of a global smart contract may exchange special messages to transfer information and value between each other if required. In this case, the (unforgeable) sender *account_id* becomes important (cf. **2.4.4**).

[13]The above needs to be literally true only for the basic workchain and its shardchains; other workchains may provide other ways of creating messages.

[14]As a degenerate case, this shardchain may coincide with the originating shardchain—for example, if we are working inside a workchain which has not yet been split.

2.4.15. Messages to any instance of a smart contract; wildcard addresses. Sometimes a message (e.g., a client request) needs be delivered to any instance of a global smart contract, usually the closest one (if there is one residing in the same shardchain as the sender, it is the obvious candidate). One way of doing this is by using a "wildcard recipient address", with the first d bits of the destination *account_id* allowed to take arbitrary values. In practice, one will usually set these d bits to the same values as in the sender's *account_id*.

2.4.16. Input queue is absent. All messages received by a blockchain (usually a shardchain; sometimes the masterchain)—or, essentially, by an "account-chain" residing inside some shardchain—are immediately delivered (i.e., processed by the receiving account). Therefore, there is no "input queue" as such. Instead, if not all messages destined for a specific shardchain can be processed because of limitations on the total size of blocks and gas usage, some messages are simply left to accumulate in the output queues of the originating shardchains.

2.4.17. Output queues. From the perspective of the Infinite Sharding Paradigm (cf. **2.1.2**), each account-chain (i.e., each account) has its own output queue, consisting of all messages it has generated, but not yet delivered to their recipients. Of course, account-chains have only a virtual existence; they are grouped into shardchains, and a shardchain has an output "queue", consisting of the union of the output queues of all accounts belonging to the shardchain.

This shardchain output "queue" imposes only partial order on its member messages. Namely, a message generated in a preceding block must be delivered before any message generated in a subsequent block, and any messages generated by the same account and having the same destination must be delivered in the order of their generation.

2.4.18. Reliable and fast inter-chain messaging. It is of paramount importance for a scalable multi-blockchain project such as TON to be able to forward and deliver messages between different shardchains (cf. **2.1.3**), even if there are millions of them in the system. The messages should be delivered *reliably* (i.e., messages should not be lost or delivered more than once) and *quickly*. The TON Blockchain achieves this goal by using a combination of two "message routing" mechanisms.

2.4.19. Hypercube routing: "slow path" for messages with assured delivery. The TON Blockchain uses "hypercube routing" as a slow, but safe and reliable way of delivering messages from one shardchain to another, using several intermediate shardchains for transit if necessary. Otherwise, the validators of any given shardchain would need to keep track of the state of (the output queues of) all other shardchains, which would require prohibitive amounts of computing power and network bandwidth as the total quantity of shardchains grows, thus limiting the scalability of the system. Therefore, it is not possible to deliver messages directly from any shard to every other. Instead, each shard is "connected" only to shards differing in exactly one hexadecimal digit of their (w, s) shard identifiers (cf. **2.1.8**). In this way, all shardchains constitute a "hypercube" graph, and messages travel along the edges of this hypercube.

If a message is sent to a shard different from the current one, one of the hexadecimal digits (chosen deterministically) of the current shard identifier is replaced by the corresponding digit of the target shard, and the resulting identifier is used as the proximate target to forward the message to.[15]

The main advantage of hypercube routing is that the block validity conditions imply that validators creating blocks of a shardchain must collect and process messages from the output queues of "neighboring" shardchains, on pain of losing their stakes. In this way, any message can be expected to reach its final destination sooner or later; a message cannot be lost in transit or delivered twice.

Notice that hypercube routing introduces some additional delays and expenses, because of the necessity to forward messages through several intermediate shardchains. However, the number of these intermediate shardchains grows very slowly, as the logarithm $\log N$ (more precisely, $\lceil \log_{16} N \rceil - 1$) of the total number of shardchains N. For example, if $N \approx 250$, there will be at most one intermediate hop; and for $N \approx 4000$ shardchains, at most two. With four intermediate hops, we can support up to one million shardchains. We think this is a very small price to pay for the essentially unlimited scalability of the system. In fact, it is not necessary to pay even this price:

2.4.20. Instant Hypercube Routing: "fast path" for messages. A novel feature of the TON Blockchain is that it introduces a "fast path" for

[15]This is not necessarily the final version of the algorithm used to compute the next hop for hypercube routing. In particular, hexadecimal digits may be replaced by r-bit groups, with r a configurable parameter, not necessarily equal to four.

forwarding messages from one shardchain to any other, allowing in most cases to bypass the "slow" hypercube routing of **2.4.19** altogether and deliver the message into the very next block of the final destination shardchain.

The idea is as follows. During the "slow" hypercube routing, the message travels (in the network) along the edges of the hypercube, but it is delayed (for approximately five seconds) at each intermediate vertex to be committed into the corresponding shardchain before continuing its voyage.

To avoid unnecessary delays, one might instead relay the message along with a suitable Merkle proof along the edges of the hypercube, without waiting to commit it into the intermediate shardchains. In fact, the network message should be forwarded from the validators of the "task group" (cf. **2.6.8**) of the original shard to the designated block producer (cf. **2.6.9**) of the "task group" of the destination shard; this might be done directly without going along the edges of the hypercube. When this message with the Merkle proof reaches the validators (more precisely, the collators; cf. **2.6.5**) of the destination shardchain, they can commit it into a new block immediately, without waiting for the message to complete its travel along the "slow path". Then a confirmation of delivery along with a suitable Merkle proof is sent back along the hypercube edges, and it may be used to stop the travel of the message along the "slow path", by committing a special transaction.

Note that this "instant delivery" mechanism does not replace the "slow" but failproof mechanism described in **2.4.19**. The "slow path" is still needed because the validators cannot be punished for losing or simply deciding not to commit the "fast path" messages into new blocks of their blockchains.[16]

Therefore, both message forwarding methods are run in parallel, and the "slow" mechanism is aborted only if a proof of success of the "fast" mechanism is committed into an intermediate shardchain.[17]

2.4.21. Collecting input messages from output queues of neighboring shardchains. When a new block for a shardchain is proposed, some of the output messages of the neighboring (in the sense of the routing hypercube of **2.4.19**) shardchains are included in the new block as "input" messages and immediately delivered (i.e., processed). There are certain rules

[16]However, the validators have some incentive to do so as soon as possible, because they will be able to collect all forwarding fees associated with the message that have not yet been consumed along the slow path.

[17]In fact, one might temporarily or permanently disable the "instant delivery" mechanism altogether, and the system would continue working, albeit more slowly.

as to the order in which these neighbors' output messages must be processed. Essentially, an "older" message (coming from a shardchain block referring to an older masterchain block) must be delivered before any "newer" message; and for messages coming from the same neighboring shardchain, the partial order of the output queue described in **2.4.17** must be observed.

2.4.22. Deleting messages from output queues. Once an output queue message is observed as having been delivered by a neighboring shardchain, it is explicitly deleted from the output queue by a special transaction.

2.4.23. Preventing double delivery of messages. To prevent double delivery of messages taken from the output queues of the neighboring shardchains, each shardchain (more precisely, each account-chain inside it) keeps the collection of recently delivered messages (or just their hashes) as part of its state. When a delivered message is observed to be deleted from the output queue by its originating neighboring shardchain (cf. **2.4.22**), it is deleted from the collection of recently delivered messages as well.

2.4.24. Forwarding messages intended for other shardchains. Hypercube routing (cf. **2.4.19**) means that sometimes outbound messages are delivered not to the shardchain containing the intended recipient, but to a neighboring shardchain lying on the hypercube path to the destination. In this case, "delivery" consists in moving the inbound message to the outbound queue. This is reflected explicitly in the block as a special *forwarding transaction*, containing the message itself. Essentially, this looks as if the message had been received by somebody inside the shardchain, and one identical message had been generated as result.

2.4.25. Payment for forwarding and keeping a message. The forwarding transaction actually spends some gas (depending on the size of the message being forwarded), so a gas payment is deducted from the value of the message being forwarded on behalf of the validators of this shardchain. This forwarding payment is normally considerably smaller than the gas payment exacted when the message is finally delivered to its recipient, even if the message has been forwarded several times because of hypercube routing. Furthermore, as long as a message is kept in the output queue of some shardchain, it is part of the shardchain's global state, so a payment for keeping global data for a long time may be also collected by special transactions.

2.4.26. Messages to and from the masterchain. Messages can be sent directly from any shardchain to the masterchain, and vice versa. However, gas prices for sending messages to and for processing messages in the masterchain are quite high, so this ability will be used only when truly necessary—for example, by the validators to deposit their stakes. In some cases, a minimal deposit (attached value) for messages sent to the masterchain may be defined, which is returned only if the message is deemed "valid" by the receiving party.

Messages cannot be automatically routed through the masterchain. A message with *workchain_id* $\neq -1$ (-1 being the special *workchain_id* indicating the masterchain) cannot be delivered to the masterchain.

In principle, one can create a message-forwarding smart contract inside the masterchain, but the price of using it would be prohibitive.

2.4.27. Messages between accounts in the same shardchain. In some cases, a message is generated by an account belonging to some shardchain, destined to another account in the same shardchain. For example, this happens in a new workchain which has not yet split into several shardchains because the load is manageable.

Such messages might be accumulated in the output queue of the shardchain and then processed as incoming messages in subsequent blocks (any shard is considered a neighbor of itself for this purpose). However, in most cases it is possible to deliver these messages within the originating block itself.

In order to achieve this, a partial order is imposed on all transactions included in a shardchain block, and the transactions (each consisting in the delivery of a message to some account) are processed respecting this partial order. In particular, a transaction is allowed to process some output message of a preceding transaction with respect to this partial order.

In this case, the message body is not copied twice. Instead, the originating and the processing transactions refer to a shared copy of the message.

2.5 Global Shardchain State. "Bag of Cells" Philosophy.

Now we are ready to describe the global state of a TON blockchain, or at least of a shardchain of the basic workchain.

We start with a "high-level" or "logical" description, which consists in saying that the global state is a value of algebraic type *ShardchainState*.

2.5.1. Shardchain state as a collection of account-chain states. According to the Infinite Sharding Paradigm (cf. **2.1.2**), any shardchain is just a (temporary) collection of virtual "account-chains", containing exactly one account each. This means that, essentially, the global shardchain state must be a hashmap

$$\mathit{ShardchainState} := (\mathit{Account} \dashrightarrow \mathit{AccountState}) \tag{23}$$

where all *account_id* appearing as indices of this hashmap must begin with prefix s, if we are discussing the state of shard (w, s) (cf. **2.1.8**).

In practice, we might want to split *AccountState* into several parts (e.g., keep the account output message queue separate to simplify its examination by the neighboring shardchains), and have several hashmaps ($\mathit{Account} \dashrightarrow \mathit{AccountStatePart}_i$) inside the *ShardchainState*. We might also add a small number of "global" or "integral" parameters to the *ShardchainState*, (e.g., the total balance of all accounts belonging to this shard, or the total number of messages in all output queues).

However, (23) is a good first approximation of what the shardchain global state looks like, at least from a "logical" ("high-level") perspective. The formal description of algebraic types *AccountState* and *ShardchainState* can be done with the aid of a TL-scheme (cf. **2.2.5**), to be provided elsewhere.

2.5.2. Splitting and merging shardchain states. Notice that the Infinite Sharding Paradigm description of the shardchain state (23) shows how this state should be processed when shards are split or merged. In fact, these state transformations turn out to be very simple operations with hashmaps.

2.5.3. Account-chain state. The (virtual) account-chain state is just the state of one account, described by type *AccountState*. Usually it has all or some of the fields listed in **2.3.20**, depending on the specific constructor used.

2.5.4. Global workchain state. Similarly to (23), we may define the global *workchain* state by the same formula, but with *account_id*'s allowed to take any values, not just those belonging to one shard. Remarks similar to those made in **2.5.1** apply in this case as well: we might want to split this hashmap into several hashmaps, and we might want to add some "integral" parameters such as the total balance.

Essentially, the global workchain state *must* be given by the same type *ShardchainState* as the shardchain state, because it is the shardchain state we would obtain if all existing shardchains of this workchain suddenly merged into one.

2.5.5. Low-level perspective: "bag of cells". There is a "low-level" description of the account-chain or shardchain state as well, complementary to the "high-level" description given above. This description is quite important, because it turns out to be pretty universal, providing a common basis for representing, storing, serializing and transferring by network almost all data used by the TON Blockchain (blocks, shardchain states, smart-contract storage, Merkle proofs, etc.). At the same time, such a universal "low-level" description, once understood and implemented, allows us to concentrate our attention on the "high-level" considerations only.

Recall that the TVM represents values of arbitrary algebraic types (including, for instance, *ShardchainState* of (23)) by means of a tree of *TVM cells*, or *cells* for short (cf. **2.3.14** and **2.2.5**).

Any such cell consists of two *descriptor bytes*, defining certain flags and values $0 \leq b \leq 128$, the quantity of raw bytes, and $0 \leq c \leq 4$, the quantity of references to other cells. Then b raw bytes and c cell references follow.[18]

The exact format of cell references depends on the implementation and on whether the cell is located in RAM, on disk, in a network packet, in a block, and so on. A useful abstract model consists in imagining that all cells are kept in content-addressable memory, with the address of a cell equal to its (SHA256) hash. Recall that the (Merkle) hash of a cell is computed exactly by replacing the references to its child cells by their (recursively computed) hashes and hashing the resulting byte string.

In this way, if we use cell hashes to reference cells (e.g., inside descriptions of other cells), the system simplifies somewhat, and the hash of a cell starts to coincide with the hash of the byte string representing it.

Now we see that *any object representable by TVM, the global shardchain state included, can be represented as a "bag of cells"*—i.e., *a collection of cells along with a "root" reference to one of them* (e.g., by hash). Notice

[18] One can show that, if Merkle proofs for all data stored in a tree of cells are needed equally often, one should use cells with $b + ch \approx 2(h + r)$ to minimize average Merkle proof size, where $h = 32$ is the hash size in bytes, and $r \approx 4$ is the "byte size" of a cell reference. In other words, a cell should contain either two references and a few raw bytes, or one reference and about 36 raw bytes, or no references at all with 72 raw bytes.

that duplicate cells are removed from this description (the "bag of cells" is a set of cells, not a multiset of cells), so the abstract tree representation might actually become a directed acyclic graph (dag) representation.

One might even keep this state on disk in a B- or B+-tree, containing all cells in question (maybe with some additional data, such as subtree height or reference counter), indexed by cell hash. However, a naive implementation of this idea would result in the state of one smart contract being scattered among distant parts of the disk file, something we would rather avoid.[19]

Now we are going to explain in some detail how almost all objects used by the TON Blockchain can be represented as "bags of cells", thus demonstrating the universality of this approach.

2.5.6. Shardchain block as a "bag of cells". A shardchain block itself can be also described by an algebraic type, and stored as a "bag of cells". Then a naive binary representation of the block may be obtained simply by concatenating the byte strings representing each of the cells in the "bag of cells", in arbitrary order. This representation might be improved and optimized, for instance, by providing a list of offsets of all cells at the beginning of the block, and replacing hash references to other cells with 32-bit indices in this list whenever possible. However, one should imagine that a block is essentially a "bag of cells", and all other technical details are just minor optimization and implementation issues.

2.5.7. Update to an object as a "bag of cells". Imagine that we have an old version of some object represented as a "bag of cells", and that we want to represent a new version of the same object, supposedly not too different from the previous one. One might simply represent the new state as another "bag of cells" with its own root, *and remove from it all cells occurring in the old version.* The remaining "bag of cells" is essentially an *update* to the object. Everybody who has the old version of this object and the update can compute the new version, simply by uniting the two bags of cells, and removing the old root (decreasing its reference counter and de-allocating the cell if the reference counter becomes zero).

[19] A better implementation would be to keep the state of the smart contract as a serialized string, if it is small, or in a separate B-tree, if it is large; then the top-level structure representing the state of a blockchain would be a B-tree, whose leaves are allowed to contain references to other B-trees.

2.5.8. Updates to the state of an account. In particular, updates to the state of an account, or to the global state of a shardchain, or to any hashmap can be represented using the idea described in **2.5.7**. This means that when we receive a new shardchain block (which is a "bag of cells"), we interpret this "bag of cells" not just by itself, but by uniting it first with the "bag of cells" representing the previous state of the shardchain. In this sense each block may "contain" the whole state of the blockchain.

2.5.9. Updates to a block. Recall that a block itself is a "bag of cells", so, if it becomes necessary to edit a block, one can similarly define a "block update" as a "bag of cells", interpreted in the presence of the "bag of cells" which is the previous version of this block. This is roughly the idea behind the "vertical blocks" discussed in **2.1.17**.

2.5.10. Merkle proof as a "bag of cells". Notice that a (generalized) Merkle proof—for example, one asserting that $x[i] = y$ starting from a known value of $\textsc{Hash}(x) = h$ (cf. **2.3.10** and **2.3.15**)—may also be represented as a "bag of cells". Namely, one simply needs to provide a subset of cells corresponding to a path from the root of $x : \mathit{Hashmap}(n, X)$ to its desired leaf with index $i : \mathbf{2}^n$ and value $y : X$. References to children of these cells not lying on this path will be left "unresolved" in this proof, represented by cell hashes. One can also provide a simultaneous Merkle proof of, say, $x[i] = y$ and $x[i'] = y'$, by including in the "bag of cells" the cells lying on the union of the two paths from the root of x to leaves corresponding to indices i and i'.

2.5.11. Merkle proofs as query responses from full nodes. In essence, a full node with a complete copy of a shardchain (or account-chain) state can provide a Merkle proof when requested by a light node (e.g., a network node running a light version of the TON Blockchain client), enabling the receiver to perform some simple queries without external help, using only the cells provided in this Merkle proof. The light node can send its queries in a serialized format to the full node, and receive the correct answers with Merkle proofs—or just the Merkle proofs, because the requester should be able to compute the answers using only the cells included in the Merkle proof. This Merkle proof would consist simply of a "bag of cells", containing only those cells belonging to the shardchain's state that have been accessed by the full node while executing the light node's query. This approach can be used in particular for executing "get queries" of smart contracts (cf. **4.3.12**).

2.5.12. Augmented update, or state update with Merkle proof of validity. Recall (cf. **2.5.7**) that we can describe the changes in an object state from an old value $x : X$ to a new value $x' : X$ by means of an "update", which is simply a "bag of cells", containing those cells that lie in the subtree representing new value x', but not in the subtree representing old value x, because the receiver is assumed to have a copy of the old value x and all its cells.

However, if the receiver does not have a full copy of x, but knows only its (Merkle) hash $h = \text{HASH}(x)$, it will not be able to check the validity of the update (i.e., that all "dangling" cell references in the update do refer to cells present in the tree of x). One would like to have "verifiable" updates, augmented by Merkle proofs of existence of all referred cells in the old state. Then anybody knowing only $h = \text{HASH}(x)$ would be able to check the validity of the update and compute the new $h' = \text{HASH}(x')$ by itself.

Because our Merkle proofs are "bags of cells" themselves (cf. **2.5.10**), one can construct such an *augmented update* as a "bag of cells", containing the old root of x, some of its descendants along with paths from the root of x to them, and the new root of x' and all its descendants that are not part of x.

2.5.13. Account state updates in a shardchain block. In particular, account state updates in a shardchain block should be augmented as discussed in **2.5.12**. Otherwise, somebody might commit a block containing an invalid state update, referring to a cell absent in the old state; proving the invalidity of such a block would be problematic (how is the challenger to prove that a cell is *not* part of the previous state?).

Now, if all state updates included in a block are augmented, their validity is easily checked, and their invalidity is also easily shown as a violation of the recursive defining property of (generalized) Merkle hashes.

2.5.14. "Everything is a bag of cells" philosophy. Previous considerations show that everything we need to store or transfer, either in the TON Blockchain or in the network, is representable as a "bag of cells". This is an important part of the TON Blockchain design philosophy. Once the "bag of cells" approach is explained and some "low-level" serializations of "bags of cells" are defined, one can simply define everything (block format, shardchain and account state, etc.) on the high level of abstract (dependent) algebraic data types.

The unifying effect of the "everything is a bag of cells" philosophy considerably simplifies the implementation of seemingly unrelated services; cf. **5.1.9**

for an example involving payment channels.

2.5.15. Block "headers" for TON blockchains. Usually, a block in a blockchain begins with a small header, containing the hash of the previous block, its creation time, the Merkle hash of the tree of all transactions contained in the block, and so on. Then the block hash is defined to be the hash of this small block header. Because the block header ultimately depends on all data included in the block, one cannot alter the block without changing its hash.

In the "bag of cells" approach used by the blocks of TON blockchains, there is no designated block header. Instead, the block hash is defined as the (Merkle) hash of the root cell of the block. Therefore, the top (root) cell of the block might be considered a small "header" of this block.

However, the root cell might not contain all the data usually expected from such a header. Essentially, one wants the header to contain some of the fields defined in the *Block* datatype. Normally, these fields will be contained in several cells, including the root. These are the cells that together constitute a "Merkle proof" for the values of the fields in question. One might insist that a block contain these "header cells" in the very beginning, before any other cells. Then one would need to download only the first several bytes of a block serialization in order to obtain all of the "header cells", and to learn all of the expected fields.

2.6 Creating and Validating New Blocks

The TON Blockchain ultimately consists of shardchain and masterchain blocks. These blocks must be created, validated and propagated through the network to all parties concerned, in order for the system to function smoothly and correctly.

2.6.1. Validators. New blocks are created and validated by special designated nodes, called *validators*. Essentially, any node wishing to become a validator may become one, provided it can deposit a sufficiently large stake (in TON coins, i.e., Grams; cf. Appendix **A**) into the masterchain. Validators obtain some "rewards" for good work, namely, the transaction, storage and gas fees from all transactions (messages) committed into newly generated blocks, and some newly minted coins, reflecting the "gratitude" of the whole community to the validators for keeping the TON Blockchain working.

This income is distributed among all participating validators proportionally to their stakes.

However, being a validator is a high responsibility. If a validator signs an invalid block, it can be punished by losing part or all of its stake, and by being temporarily or permanently excluded from the set of validators. If a validator does not participate in creating a block, it does not receive its share of the reward associated with that block. If a validator abstains from creating new blocks for a long time, it may lose part of its stake and be suspended or permanently excluded from the set of validators.

All this means that the validator does not get its money "for nothing". Indeed, it must keep track of the states of all or some shardchains (each validator is responsible for validating and creating new blocks in a certain subset of shardchains), perform all computations requested by smart contracts in these shardchains, receive updates about other shardchains and so on. This activity requires considerable disk space, computing power and network bandwidth.

2.6.2. Validators instead of miners. Recall that the TON Blockchain uses the Proof-of-Stake approach, instead of the Proof-of-Work approach adopted by Bitcoin, the current version of Ethereum, and most other cryptocurrencies. This means that one cannot "mine" a new block by presenting some proof-of-work (computing a lot of otherwise useless hashes) and obtain some new coins as a result. Instead, one must become a validator and spend one's computing resources to store and process TON Blockchain requests and data. In short, *one must be a validator to mine new coins.* In this respect, *validators are the new miners.*

However, there are some other ways to earn coins apart from being a validator.

2.6.3. Nominators and "mining pools". To become a validator, one would normally need to buy and install several high-performance servers and acquire a good Internet connection for them. This is not so expensive as the ASIC equipment currently required to mine Bitcoins. However, one definitely cannot mine new TON coins on a home computer, let alone a smartphone.

In the Bitcoin, Ethereum and other Proof-of-Work cryptocurrency mining communities there is a notion of *mining pools*, where a lot of nodes, having insufficient computing power to mine new blocks by themselves, combine their efforts and share the reward afterwards.

A corresponding notion in the Proof-of-Stake world is that of a *nominator*. Essentially, this is a node lending its money to help a validator increase its stake; the validator then distributes the corresponding share of its reward (or some previously agreed fraction of it—say, 50%) to the nominator.

In this way, a nominator can also take part in the "mining" and obtain some reward proportional to the amount of money it is willing to deposit for this purpose. It receives only a fraction of the corresponding share of the validator's reward, because it provides only the "capital", but does not need to buy computing power, storage and network bandwidth.

However, if the validator loses its stake because of invalid behavior, the nominator loses its share of the stake as well. In this sense the nominator *shares the risk*. It must choose its nominated validator wisely, otherwise it can lose money. In this sense, nominators make a weighted decision and "vote" for certain validators with their funds.

On the other hand, this nominating or lending system enables one to become a validator without investing a large amount of money into Grams (TON coins) first. In other words, it prevents those keeping large amounts of Grams from monopolizing the supply of validators.

2.6.4. Fishermen: obtaining money by pointing out others' mistakes. Another way to obtain some rewards without being a validator is by becoming a *fisherman*. Essentially, any node can become a fisherman by making a small deposit in the masterchain. Then it can use special masterchain transactions to publish (Merkle) invalidity proofs of some (usually shardchain) blocks previously signed and published by validators. If other validators agree with this invalidity proof, the offending validators are punished (by losing part of their stake), and the fisherman obtains some reward (a fraction of coins confiscated from the offending validators). Afterwards, the invalid (shardchain) block must be corrected as outlined in **2.1.17**. Correcting invalid masterchain blocks may involve creating "vertical" blocks on top of previously committed masterchain blocks (cf. **2.1.17**); there is no need to create a fork of the masterchain.

Normally, a fisherman would need to become a full node for at least some shardchains, and spend some computing resources by running the code of at least some smart contracts. While a fisherman does not need to have as much computing power as a validator, we think that a natural candidate to become a fisherman is a would-be validator that is ready to process new blocks, but has not yet been elected as a validator (e.g., because of a failure

to deposit a sufficiently large stake).

2.6.5. Collators: obtaining money by suggesting new blocks to validators. Yet another way to obtain some rewards without being a validator is by becoming a *collator*. This is a node that prepares and suggests to a validator new shardchain block candidates, complemented (collated) with data taken from the state of this shardchain and from other (usually neighboring) shardchains, along with suitable Merkle proofs. (This is necessary, for example, when some messages need to be forwarded from neighboring shardchains.) Then a validator can easily check the proposed block candidate for validity, without having to download the complete state of this or other shardchains.

Because a validator needs to submit new (collated) block candidates to obtain some ("mining") rewards, it makes sense to pay some part of the reward to a collator willing to provide suitable block candidates. In this way, a validator may free itself from the necessity of watching the state of the neighboring shardchains, by outsourcing it to a collator.

However, we expect that during the system's initial deployment phase there will be no separate designated collators, because all validators will be able to act as collators for themselves.

2.6.6. Collators or validators: obtaining money for including user transactions. Users can open micropayment channels to some collators or validators and pay small amounts of coins in exchange for the inclusion of their transactions in the shardchain.

2.6.7. Global validator set election. The "global" set of validators is elected once each month (actually, every 2^{19} masterchain blocks). This set is determined and universally known one month in advance.

In order to become a validator, a node must transfer some TON coins (Grams) into the masterchain, and then send them to a special smart contract as its suggested stake s. Another parameter, sent along with the stake, is $l \geq 1$, the maximum validating load this node is willing to accept relative to the minimal possible. There is also a global upper bound (another configurable parameter) L on l, equal to, say, 10.

Then the global set of validators is elected by this smart contract, simply by selecting up to T candidates with maximal suggested stakes and publishing their identities. Originally, the total number of validators is $T = 100$; we

expect it to grow to 1000 as the load increases. It is a configurable parameter (cf. **2.1.21**).

The actual stake of each validator is computed as follows: If the top T proposed stakes are $s_1 \geq s_2 \geq \cdots \geq s_T$, the actual stake of i-th validator is set to $s'_i := \min(s_i, l_i \cdot s_T)$. In this way, $s'_i/s'_T \leq l_i$, so the i-th validator does not obtain more than $l_i \leq L$ times the load of the weakest validator (because the load is ultimately proportional to the stake).

Then elected validators may withdraw the unused part of their stake, $s_i - s'_i$. Unsuccessful validator candidates may withdraw all of their proposed stake.

Each validator publishes its *public signing key*, not necessarily equal to the public key of the account the stake came from.[20]

The stakes of the validators are frozen until the end of the period for which they have been elected, and one month more, in case new disputes arise (i.e., an invalid block signed by one of these validators is found). After that, the stake is returned, along with the validator's share of coins minted and fees from transactions processed during this time.

2.6.8. Election of validator "task groups". The whole global set of validators (where each validator is considered present with multiplicity equal to its stake—otherwise a validator might be tempted to assume several identities and split its stake among them) is used only to validate new masterchain blocks. The shardchain blocks are validated only by specially selected subsets of validators, taken from the global set of validators chosen as described in **2.6.7**.

These validator "subsets" or "task groups", defined for every shard, are rotated each hour (actually, every 2^{10} masterchain blocks), and they are known one hour in advance, so that every validator knows which shards it will need to validate, and can prepare for that (e.g., by downloading missing shardchain data).

The algorithm used to select validator task groups for each shard (w, s) is deterministic pseudorandom. It uses pseudorandom numbers embedded by validators into each masterchain block (generated by a consensus using threshold signatures) to create a random seed, and then computes for example $\text{HASH}(\text{CODE}(w).\,\text{CODE}(s).\mathit{validator\_id}.\mathit{rand\_seed})$ for each validator. Then validators are sorted by the value of this hash, and the first several are

[20] It makes sense to generate and use a new key pair for every validator election.

selected, so as to have at least $20/T$ of the total validator stakes and consist of at least 5 validators.

This selection could be done by a special smart contract. In that case, the selection algorithm would easily be upgradable without hard forks by the voting mechanism mentioned in **2.1.21**. All other "constants" mentioned so far (such as 2^{19}, 2^{10}, T, 20, and 5) are also configurable parameters.

2.6.9. Rotating priority order on each task group. There is a certain "priority" order imposed on the members of a shard task group, depending on the hash of the previous masterchain block and (shardchain) block sequence number. This order is determined by generating and sorting some hashes as described above.

When a new shardchain block needs to be generated, the shard task group validator selected to create this block is normally the first one with respect to this rotating "priority" order. If it fails to create the block, the second or third validator may do it. Essentially, all of them may suggest their block candidates, but the candidate suggested by the validator having the highest priority should win as the result of Byzantine Fault Tolerant (BFT) consensus protocol.

2.6.10. Propagation of shardchain block candidates. Because shardchain task group membership is known one hour in advance, their members can use that time to build a dedicated "shard validators multicast overlay network", using the general mechanisms of the TON Network (cf. **3.3**). When a new shardchain block needs to be generated—normally one or two seconds after the most recent masterchain block has been propagated—everybody knows who has the highest priority to generate the next block (cf. **2.6.9**). This validator will create a new collated block candidate, either by itself or with the aid of a collator (cf. **2.6.5**). The validator must check (validate) this block candidate (especially if it has been prepared by some collator) and sign it with its (validator) private key. Then the block candidate is propagated to the remainder of the task group using the prearranged multicast overlay network (the task group creates its own private overlay network as explained in **3.3**, and then uses a version of the streaming multicast protocol described in **3.3.15** to propagate block candidates).

A truly BFT way of doing this would be to use a Byzantine multicast protocol, such as the one used in Honey Badger BFT [11]: encode the block candidate by an $(N, 2N/3)$-erasure code, send $1/N$ of the resulting data

directly to each member of the group, and expect them to multicast directly their part of the data to all other members of the group.

However, a faster and more straightforward way of doing this (cf. also **3.3.15**) is to split the block candidate into a sequence of signed one-kilobyte blocks ("chunks"), augment their sequence by a Reed–Solomon or a fountain code (such as the RaptorQ code [9] [14]), and start transmitting chunks to the neighbors in the "multicast mesh" (i.e., the overlay network), expecting them to propagate these chunks further. Once a validator obtains enough chunks to reconstruct the block candidate from them, it signs a confirmation receipt and propagates it through its neighbors to the whole of the group. Then its neighbors stop sending new chunks to it, but may continue to send the (original) signatures of these chunks, believing that this node can generate the subsequent chunks by applying the Reed–Solomon or fountain code by itself (having all data necessary), combine them with signatures, and propagate to its neighbors that are not yet ready.

If the "multicast mesh" (overlay network) remains connected after removing all "bad" nodes (recall that up to one-third of nodes are allowed to be bad in a Byzantine way, i.e., behave in arbitrary malicious fashion), this algorithm will propagate the block candidate as quickly as possible.

Not only the designated high-priority block creator may multicast its block candidate to the whole of the group. The second and third validator by priority may start multicasting their block candidates, either immediately or after failing to receive a block candidate from the top priority validator. However, normally only the block candidate with maximal priority will be signed by all (actually, by at least two-thirds of the task group) validators and committed as a new shardchain block.

2.6.11. Validation of block candidates. Once a block candidate is received by a validator and the signature of its originating validator is checked, the receiving validator checks the validity of this block candidate, by performing all transactions in it and checking that their result coincides with the one claimed. All messages imported from other blockchains must be supported by suitable Merkle proofs in the collated data, otherwise the block candidate is deemed invalid (and, if a proof of this is committed to the masterchain, the validators having already signed this block candidate may be punished). On the other hand, if the block candidate is found to be valid, the receiving validator signs it and propagates its signature to other validators in the group, either through the "mesh multicast network", or by direct

network messages.

We would like to emphasize that *a validator does not need access to the states of this or neighboring shardchains in order to check the validity of a (collated) block candidate*.[21] This allows the validation to proceed very quickly (without disk accesses), and lightens the computational and storage burden on the validators (especially if they are willing to accept the services of outside collators for creating block candidates).

2.6.12. Election of the next block candidate. Once a block candidate collects at least two-thirds (by stake) of the validity signatures of validators in the task group, it is eligible to be committed as the next shardchain block. A BFT protocol is run to achieve consensus on the block candidate chosen (there may be more than one proposed), with all "good" validators preferring the block candidate with the highest priority for this round. As a result of running this protocol, the block is augmented by signatures of at least two-thirds of the validators (by stake). These signatures testify not only to the validity of the block in question, but also to its being elected by the BFT protocol. After that, the block (without collated data) is combined with these signatures, serialized in a deterministic way, and propagated through the network to all parties concerned.

2.6.13. Validators must keep the blocks they have signed. During their membership in the task group and for at least one hour (or rather 2^{10} blocks) afterward, the validators are expected to keep the blocks they have signed and committed. The failure to provide a signed block to other validators may be punished.

2.6.14. Propagating the headers and signatures of new shardchain blocks to all validators. Validators propagate the headers and signatures of newly-generated shardchain blocks to the *global* set of validators, using a multicast mesh network similar to the one created for each task group.

2.6.15. Generation of new masterchain blocks. After all (or almost all) new shardchain blocks have been generated, a new masterchain block may be generated. The procedure is essentially the same as for shardchain blocks (cf. **2.6.12**), with the difference that *all* validators (or at least two-thirds of

[21] A possible exception is the state of output queues of the neighboring shardchains, needed to guarantee the message ordering requirements described in **2.4.21**, because the size of Merkle proofs might become prohibitive in this case.

them) must participate in this process. Because the headers and signatures of new shardchain blocks are propagated to all validators, hashes of the newest blocks in each shardchain can and must be included in the new masterchain block. Once these hashes are committed into the masterchain block, outside observers and other shardchains may consider the new shardchain blocks committed and immutable (cf. **2.1.13**).

2.6.16. Validators must keep the state of masterchain. A noteworthy difference between the masterchain and the shardchains is that all validators are expected to keep track of the masterchain state, without relying on collated data. This is important because the knowledge of validator task groups is derived from the masterchain state.

2.6.17. Shardchain blocks are generated and propagated in parallel. Normally, each validator is a member of several shardchain task groups; their quantity (hence the load on the validator) is approximately proportional to the validator's stake. This means that the validator runs several instances of new shardchain block generation protocol in parallel.

2.6.18. Mitigation of block retention attacks. Because the total set of validators inserts a new shardchain block's hash into the masterchain after having seen only its header and signatures, there is a small probability that the validators that have generated this block will conspire and try to avoid publishing the new block in its entirety. This would result in the inability of validators of neighboring shardchains to create new blocks, because they must know at least the output message queue of the new block, once its hash has been committed into the masterchain.

In order to mitigate this, the new block must collect signatures from some other validators (e.g., two-thirds of the union of task groups of neighboring shardchains) testifying that these validators do have copies of this block and are willing to send them to any other validators if required. Only after these signatures are presented may the new block's hash be included in the masterchain.

2.6.19. Masterchain blocks are generated later than shardchain blocks. Masterchain blocks are generated approximately once every five seconds, as are shardchain blocks. However, while the generation of new blocks in all shardchains runs essentially at the same time (normally triggered by the release of a new masterchain block), the generation of new

masterchain blocks is deliberately delayed, to allow the inclusion of hashes of newly-generated shardchain blocks in the masterchain.

2.6.20. Slow validators may receive lower rewards. If a validator is "slow", it may fail to validate new block candidates, and two-thirds of the signatures required to commit the new block may be gathered without its participation. In this case, it will receive a lower share of the reward associated with this block.

This provides an incentive for the validators to optimize their hardware, software, and network connection in order to process user transactions as fast as possible.

However, if a validator fails to sign a block before it is committed, its signature may be included in one of the next blocks, and then a part of the reward (exponentially decreasing depending on how many blocks have been generated since—e.g., 0.9^k if the validator is k blocks late) will be still given to this validator.

2.6.21. "Depth" of validator signatures. Normally, when a validator signs a block, the signature testifies only to the *relative validity* of a block: this block is valid provided all previous blocks in this and other shardchains are valid. The validator cannot be punished for taking for granted invalid data committed into previous blocks.

However, the validator signature of a block has an integer parameter called "depth". If it is non-zero, it means that the validator asserts the (relative) validity of the specified number of previous blocks as well. This is a way for "slow" or "temporarily offline" validators to catch up and sign some of the blocks that have been committed without their signatures. Then some part of the block reward will still be given to them (cf. **2.6.20**).

2.6.22. Validators are responsible for *relative* validity of signed shardchain blocks; absolute validity follows. We would like to emphasize once again that a validator's signature on a shardchain block B testifies to only the *relative* validity of that block (or maybe of d previous blocks as well, if the signature has "depth" d, cf. **2.6.21**; but this does not affect the following discussion much). In other words, the validator asserts that the next state s' of the shardchain is obtained from the previous state s by applying the block evaluation function *ev_block* described in **2.2.6**:

$$s' = ev\_block(B)(s) \tag{24}$$

In this way, the validator that signed block B cannot be punished if the original state s turns out to be "incorrect" (e.g., because of the invalidity of one of the previous blocks). A fisherman (cf. **2.6.4**) should complain only if it finds a block that is *relatively* invalid. The PoS system as a whole endeavors to make every block *relatively* valid, not *recursively (or absolutely)* valid. Notice, however, that *if all blocks in a blockchain are relatively valid, then all of them and the blockchain as a whole are absolutely valid*; this statement is easily shown using mathematical induction on the length of the blockchain. In this way, easily verifiable assertions of *relative* validity of blocks together demonstrate the much stronger *absolute validity* of the whole blockchain.

Note that by signing a block B the validator asserts that the block is valid given the original state s (i.e., that the result of (24) is not the value $\perp$ indicating that the next state cannot be computed). In this way, the validator must perform minimal formal checks of the cells of the original state that are accessed during the evaluation of (24).

For example, imagine that the cell expected to contain the original balance of an account accessed from a transaction committed into a block turns out to have zero raw bytes instead of the expected 8 or 16. Then the original balance simply cannot be retrieved from the cell, and an "unhandled exception" happens while trying to process the block. In this case, the validator should not sign such a block on pain of being punished.

2.6.23. Signing masterchain blocks. The situation with the masterchain blocks is somewhat different: by signing a masterchain block, the validator asserts not only its relative validity, but also the relative validity of all preceding blocks up to the very first block when this validator assumed its responsibility (but not further back).

2.6.24. The total number of validators. The upper limit T for the total number of validators to be elected (cf. **2.6.7**) cannot become, in the system described so far, more than, say, several hundred or a thousand, because all validators are expected to participate in a BFT consensus protocol to create each new masterchain block, and it is not clear whether such protocols can scale to thousands of participants. Even more importantly, masterchain blocks must collect the signatures of at least two-thirds of all the validators (by stake), and these signatures must be included in the new block (otherwise all other nodes in the system would have no reason to trust the new block without validating it by themselves). If more than, say, one thousand validator signatures would have to be included in each masterchain block,

this would imply more data in each masterchain block, to be stored by all full nodes and propagated through the network, and more processing power spent to check these signatures (in a PoS system, full nodes do not need to validate blocks by themselves, but they need to check the validators' signatures instead).

While limiting T to a thousand validators seems more than sufficient for the first phase of the deployment of the TON Blockchain, a provision must be made for future growth, when the total number of shardchains becomes so large that several hundred validators will not suffice to process all of them. To this end, we introduce an additional configurable parameter $T' \leq T$ (originally equal to T), and only the top T' elected validators (by stake) are expected to create and sign new masterchain blocks.

2.6.25. Decentralization of the system. One might suspect that a Proof-of-Stake system such as the TON Blockchain, relying on $T \approx 1000$ validators to create all shardchain and masterchain blocks, is bound to become "too centralized", as opposed to conventional Proof-of-Work blockchains like Bitcoin or Ethereum, where everybody (in principle) might mine a new block, without an explicit upper limit on the total number of miners.

However, popular Proof-of-Work blockchains, such as Bitcoin and Ethereum, currently require vast amounts of computing power (high "hash rates") to mine new blocks with non-negligible probability of success. Thus, the mining of new blocks tends to be concentrated in the hands of several large players, who invest huge amounts money into datacenters filled with custom-designed hardware optimized for mining; and in the hands of several large mining pools, which concentrate and coordinate the efforts of larger groups of people who are not able to provide a sufficient "hash rate" by themselves.

Therefore, as of 2017, more than 75% of new Ethereum or Bitcoin blocks are produced by less than ten miners. In fact, the two largest Ethereum mining pools produce together more than half of all new blocks! Clearly, such a system is much more centralized than one relying on $T \approx 1000$ nodes to produce new blocks.

One might also note that the investment required to become a TON Blockchain validator—i.e., to buy the hardware (say, several high-performance servers) and the stake (which can be easily collected through a pool of nominators if necessary; cf. **2.6.3**)—is much lower than that required to become a successful stand-alone Bitcoin or Ethereum miner. In fact, the parameter L of **2.6.7** will force nominators not to join the largest "mining pool"

(i.e., the validator that has amassed the largest stake), but rather to look for smaller validators currently accepting funds from nominators, or even to create new validators, because this would allow a higher proportion s'_i/s_i of the validator's—and by extension also the nominator's—stake to be used, hence yielding larger rewards from mining. In this way, the TON Proof-of-Stake system actually *encourages* decentralization (creating and using more validators) and *punishes* centralization.

2.6.26. Relative reliability of a block. The *(relative) reliability* of a block is simply the total stake of all validators that have signed this block. In other words, this is the amount of money certain actors would lose if this block turns out to be invalid. If one is concerned with transactions transferring value lower than the reliability of the block, one can consider them to be safe enough. In this sense, the relative reliability is a measure of trust an outside observer can have in a particular block.

Note that we speak of the *relative* reliability of a block, because it is a guarantee that the block is valid *provided the previous block and all other shardchains' blocks referred to are valid* (cf. **2.6.22**).

The relative reliability of a block can grow after it is committed—for example, when belated validators' signatures are added (cf. **2.6.21**). On the other hand, if one of these validators loses part or all of its stake because of its misbehavior related to some other blocks, the relative reliability of a block may *decrease*.

2.6.27. "Strengthening" the blockchain. It is important to provide incentives for validators to increase the relative reliability of blocks as much as possible. One way of doing this is by allocating a small reward to validators for adding signatures to blocks of other shardchains. Even "would-be" validators, who have deposited a stake insufficient to get into the top T validators by stake and to be included in the global set of validators (cf. **2.6.7**), might participate in this activity (if they agree to keep their stake frozen instead of withdrawing it after having lost the election). Such would-be validators might double as fishermen (cf. **2.6.4**): if they have to check the validity of certain blocks anyway, they might as well opt to report invalid blocks and collect the associated rewards.

2.6.28. Recursive reliability of a block. One can also define the *recursive reliability* of a block to be the minimum of its relative reliability and the recursive reliabilities of all blocks it refers to (i.e., the masterchain block, the

previous shardchain block, and some blocks of neighboring shardchains). In other words, if the block turns out to be invalid, either because it is invalid by itself or because one of the blocks it depends on is invalid, then at least this amount of money would be lost by someone. If one is truly unsure whether to trust a specific transaction in a block, one should compute the *recursive* reliability of this block, not just the *relative* one.

It does not make sense to go too far back when computing recursive reliability, because, if we look too far back, we will see blocks signed by validators whose stakes have already been unfrozen and withdrawn. In any case, we do not allow the validators to automatically reconsider blocks that are that old (i.e., created more than two months ago, if current values of configurable parameters are used), and create forks starting from them or correct them with the aid of "vertical blockchains" (cf. **2.1.17**), even if they turn out to be invalid. We assume that a period of two months provides ample opportunities for detecting and reporting any invalid blocks, so that if a block is not challenged during this period, it is unlikely to be challenged at all.

2.6.29. Consequence of Proof-of-Stake for light nodes. An important consequence of the Proof-of-Stake approach used by the TON Blockchain is that a light node (running light client software) for the TON Blockchain does not need to download the "headers" of all shardchain or even masterchain blocks in order to be able to check by itself the validity of Merkle proofs provided to it by full nodes as answers to its queries.

Indeed, because the most recent shardchain block hashes are included in the masterchain blocks, a full node can easily provide a Merkle proof that a given shardchain block is valid starting from a known hash of a masterchain block. Next, the light node needs to know only the very first block of the masterchain (where the very first set of validators is announced), which (or at least the hash of which) might be built-in into the client software, and only one masterchain block approximately every month afterwards, where newly-elected validator sets are announced, because this block will have been signed by the previous set of validators. Starting from that, it can obtain the several most recent masterchain blocks, or at least their headers and validator signatures, and use them as a base for checking Merkle proofs provided by full nodes.

2.7 Splitting and Merging Shardchains

One of the most characteristic and unique features of the TON Blockchain is its ability to automatically split a shardchain in two when the load becomes too high, and merge them back if the load subsides (cf. **2.1.10**). We must discuss it in some detail because of its uniqueness and its importance to the scalability of the whole project.

2.7.1. Shard configuration. Recall that, at any given moment of time, each workchain w is split into one or several shardchains (w, s) (cf. **2.1.8**). These shardchains may be represented by leaves of a binary tree, with root $(w, \emptyset)$, and each non-leaf node (w, s) having children $(w, s.0)$ and $(w, s.1)$. In this way, every account belonging to workchain w is assigned to exactly one shard, and everybody who knows the current shardchain configuration can determine the shard (w, s) containing account *account_id*: it is the only shard with binary string s being a prefix of *account_id*.

The shard configuration—i.e., this *shard binary tree*, or the collection of all active (w, s) for a given w (corresponding to the leaves of the shard binary tree)—is part of the masterchain state and is available to everybody who keeps track of the masterchain.[22]

2.7.2. Most recent shard configuration and state. Recall that hashes of the most recent shardchain blocks are included in each masterchain block. These hashes are organized in a shard binary tree (actually, a collection of trees, one for each workchain). In this way, each masterchain block contains the most recent shard configuration.

2.7.3. Announcing and performing changes in the shard configuration. The shard configuration may be changed in two ways: either a shard (w, s) can be *split* into two shards $(w, s.0)$ and $(w, s.1)$, or two "sibling" shards $(w, s.0)$ and $(w, s.1)$ can be *merged* into one shard (w, s).

These split/merge operations are announced several (e.g., 2^6; this is a configurable parameter) blocks in advance, first in the "headers" of the corresponding shardchain blocks, and then in the masterchain block that refers to these shardchain blocks. This advance announcement is needed for all parties concerned to prepare for the planned change (e.g., build an overlay multicast network to distribute new blocks of the newly-created shardchains,

[22] Actually, the shard configuration is completely determined by the last masterchain block; this simplifies getting access to the shard configuration.

as discussed in **3.3**). Then the change is committed, first into the (header of the) shardchain block (in case of a split; for a merge, blocks of both shardchains should commit the change), and then propagated to the masterchain block. In this way, the masterchain block defines not only the most recent shard configuration *before* its creation, but also the next immediate shard configuration.

2.7.4. Validator task groups for new shardchains. Recall that each shard, i.e., each shardchain, normally is assigned a subset of validators (a validator task group) dedicated to creating and validating new blocks in the corresponding shardchain (cf. **2.6.8**). These task groups are elected for some period of time (approximately one hour) and are known some time in advance (also approximately one hour), and are immutable during this period.[23]

However, the actual shard configuration may change during this period because of split/merge operations. One must assign task groups to newly created shards. This is done as follows:

Notice that any active shard (w, s) will either be a descendant of some uniquely determined original shard (w, s'), meaning that s' is a prefix of s, or it will be the root of a subtree of original shards (w, s'), where s will be a prefix of every s'. In the first case, we simply take the task group of the original shard (w, s') to double as the task group of the new shard (w, s). In the latter case, the task group of the new shard (w, s) will be the union of task groups of all original shards (w, s') that are descendants of (w, s) in the shard tree.

In this way, every active shard (w, s) gets assigned a well-defined subset of validators (task group). When a shard is split, both children inherit the whole of the task group from the original shard. When two shards are merged, their task groups are also merged.

Anyone who keeps track of the masterchain state can compute validator task groups for each of the active shards.

2.7.5. Limit on split/merge operations during the period of responsibility of original task groups. Ultimately, the new shard configuration will be taken into account, and new dedicated validator subsets (task groups) will automatically be assigned to each shard. Before that happens, one must impose a certain limit on split/merge operations; otherwise, an original task

[23]Unless some validators are temporarily or permanently banned because of signing invalid blocks—then they are automatically excluded from all task groups.

group may end up validating 2^k shardchains for a large k at the same time, if the original shard quickly splits into 2^k new shards.

This is achieved by imposing limits on how far the active shard configuration may be removed from the original shard configuration (the one used to select validator task groups currently in charge). For example, one might require that the distance in the shard tree from an active shard (w, s) to an original shard (w, s') must not exceed 3, if s' is a predecessor of s (i.e., s' is a prefix of binary string s), and must not exceed 2, if s' is a successor of s (i.e., s is a prefix of s'). Otherwise, the split or merge operation is not permitted.

Roughly speaking, one is imposing a limit on the number of times a shard can be split (e.g., three) or merged (e.g., two) during the period of responsibility of a given collection of validator task groups. Apart from that, after a shard has been created by merging or splitting, it cannot be reconfigured for some period of time (some number of blocks).

2.7.6. Determining the necessity of split operations. The split operation for a shardchain is triggered by certain formal conditions (e.g., if for 64 consecutive blocks the shardchain blocks are at least 90% full). These conditions are monitored by the shardchain task group. If they are met, first a "split preparation" flag is included in the header of a new shardchain block (and propagated to the masterchain block referring to this shardchain block). Then, several blocks afterwards, the "split commit" flag is included in the header of the shardchain block (and propagated to the next masterchain block).

2.7.7. Performing split operations. After the "split commit" flag is included in a block B of shardchain (w, s), there cannot be a subsequent block B' in that shardchain. Instead, two blocks B'_0 and B'_1 of shardchains $(w, s.0)$ and $(w, s.1)$, respectively, will be created, both referring to block B as their previous block (and both of them will indicate by a flag in the header that the shard has been just split). The next masterchain block will contain hashes of blocks B'_0 and B'_1 of the new shardchains; it is not allowed to contain the hash of a new block B' of shardchain (w, s), because a "split commit" event has already been committed into the previous masterchain block.

Notice that both new shardchains will be validated by the same validator task group as the old one, so they will automatically have a copy of their state. The state splitting operation itself is quite simple from the perspective of the Infinite Sharding Paradigm (cf. **2.5.2**).

2.7.8. Determining the necessity of merge operations. The necessity of shard merge operations is also detected by certain formal conditions (e.g., if for 64 consecutive blocks the sum of the sizes of the two blocks of sibling shardchains does not exceed 60% of maximal block size). These formal conditions should also take into account the total gas spent by these blocks and compare it to the current block gas limit, otherwise the blocks may happen to be small because there are some computation-intensive transactions that prevent the inclusion of more transactions.

These conditions are monitored by validator task groups of both sibling shards $(w, s.0)$ and $(w, s.1)$. Notice that siblings are necessarily neighbors with respect to hypercube routing (cf. **2.4.19**), so validators from the task group of any shard will be monitoring the sibling shard to some extent anyways.

When these conditions are met, either one of the validator subgroups can suggest to the other that they merge by sending a special message. Then they combine into a provisional "merged task group", with combined membership, capable of running BFT consensus algorithms and of propagating block updates and block candidates if necessary.

If they reach consensus on the necessity and readiness of merging, "merge prepare" flags are committed into the headers of some blocks of each shardchain, along with the signatures of at least two-thirds of the validators of the sibling's task group (and are propagated to the next masterchain blocks, so that everybody can get ready for the imminent reconfiguration). However, they continue to create separate shardchain blocks for some predefined number of blocks.

2.7.9. Performing merge operations. After that, when the validators from the union of the two original task groups are ready to become validators for the merged shardchain (this might involve a state transfer from the sibling shardchain and a state merge operation), they commit a "merge commit" flag in the headers of blocks of their shardchain (this event is propagated to the next masterchain blocks), and stop creating new blocks in separate shardchains (once the merge commit flag appears, creating blocks in separate shardchains is forbidden). Instead, a merged shardchain block is created (by the union of the two original task groups), referring to both of its "preceding blocks" in its "header". This is reflected in the next masterchain block, which will contain the hash of the newly created block of the merged shardchain. After that, the merged task group continues creating blocks in the merged

shardchain.

2.8 Classification of Blockchain Projects

We will conclude our brief discussion of the TON Blockchain by comparing it with existing and proposed blockchain projects. Before doing this, however, we must introduce a sufficiently general classification of blockchain projects. The comparison of particular blockchain projects, based on this classification, is postponed until **2.9**.

2.8.1. Classification of blockchain projects. As a first step, we suggest some classification criteria for blockchains (i.e., for blockchain projects). Any such classification is somewhat incomplete and superficial, because it must ignore some of the most specific and unique features of the projects under consideration. However, we feel that this is a necessary first step in providing at least a very rough and approximate map of the blockchain projects territory.

The list of criteria we consider is the following:

- Single-blockchain vs. multi-blockchain architecture (cf. **2.8.2**)
- Consensus algorithm: Proof-of-Stake vs. Proof-of-Work (cf. **2.8.3**)
- For Proof-of-Stake systems, the exact block generation, validation and consensus algorithm used (the two principal options are DPOS vs. BFT; cf. **2.8.4**)
- Support for "arbitrary" (Turing-complete) smart contracts (cf. **2.8.6**)

Multi-blockchain systems have additional classification criteria (cf. **2.8.7**):

- Type and rules of member blockchains: homogeneous, heterogeneous (cf. **2.8.8**), mixed (cf. **2.8.9**). Confederations (cf. **2.8.10**).
- Absence or presence of a *masterchain*, internal or external (cf. **2.8.11**)
- Native support for sharding (cf. **2.8.12**). Static or dynamic sharding (cf. **2.8.13**).
- Interaction between member blockchains: loosely-coupled and tightly-coupled systems (cf. **2.8.14**)

2.8.2. Single-blockchain vs. multi-blockchain projects. The first classification criterion is the quantity of blockchains in the system. The oldest and simplest projects consist of a *single blockchain* ("singlechain projects" for short); more sophisticated projects use (or, rather, plan to use) *multiple blockchains* ("multichain projects").

Singlechain projects are generally simpler and better tested; they have withstood the test of time. Their main drawback is low performance, or at least transaction throughput, which is on the level of ten (Bitcoin) to less than one hundred[24] (Ethereum) transactions per second for general-purpose systems. Some specialized systems (such as Bitshares) are capable of processing tens of thousands of specialized transactions per second, at the expense of requiring the blockchain state to fit into memory, and limiting the processing to a predefined special set of transactions, which are then executed by highly-optimized code written in languages like C++ (no VMs here).

Multichain projects promise the scalability everybody craves. They may support larger total states and more transactions per second, at the expense of making the project much more complex, and its implementation more challenging. As a result, there are few multichain projects already running, but most proposed projects are multichain. We believe that the future belongs to multichain projects.

2.8.3. Creating and validating blocks: Proof-of-Work vs. Proof-of-Stake. Another important distinction is the algorithm and protocol used to create and propagate new blocks, check their validity, and select one of several forks if they appear.

The two most common paradigms are *Proof-of-Work (PoW)* and *Proof-of-Stake (PoS)*. The Proof-of-Work approach usually allows any node to create ("mine") a new block (and obtain some reward associated with mining a block) if it is lucky enough to solve an otherwise useless computational problem (usually involving the computation of a large amount of hashes) before other competitors manage to do this. In the case of forks (for example, if two nodes publish two otherwise valid but different blocks to follow the previous one) the longest fork wins. In this way, the guarantee of immutability of the blockchain is based on the amount of *work* (computational resources) spent to generate the blockchain: anybody who would like to create a fork of this blockchain would need to re-do this work to create alternative versions of the

[24]More like 15, for the time being. However, some upgrades are being planned to make Ethereum transaction throughput several times larger.

already committed blocks. For this, one would need to control more than 50% of the total computing power spent creating new blocks, otherwise the alternative fork will have exponentially low chances of becoming the longest.

The Proof-of-Stake approach is based on large *stakes* (nominated in cryptocurrency) made by some special nodes (*validators*) to assert that they have checked (*validated*) some blocks and have found them correct. Validators sign blocks, and receive some small rewards for this; however, if a validator is ever caught signing an incorrect block, and a proof of this is presented, part or all of its stake is forfeit. In this way, the guarantee of validity and immutability of the blockchain is given by the total volume of stakes put by validators on the validity of the blockchain.

The Proof-of-Stake approach is more natural in the respect that it incentivizes the validators (which replace PoW miners) to perform useful computation (needed to check or create new blocks, in particular, by performing all transactions listed in a block) instead of computing otherwise useless hashes. In this way, validators would purchase hardware that is better adapted to processing user transactions, in order to receive rewards associated with these transactions, which seems quite a useful investment from the perspective of the system as a whole.

However, Proof-of-Stake systems are somewhat more challenging to implement, because one must provide for many rare but possible conditions. For example, some malicious validators might conspire to disrupt the system to extract some profit (e.g., by altering their own cryptocurrency balances). This leads to some non-trivial game-theoretic problems.

In short, Proof-of-Stake is more natural and more promising, especially for multichain projects (because Proof-of-Work would require prohibitive amounts of computational resources if there are many blockchains), but must be more carefully thought out and implemented. Most currently running blockchain projects, especially the oldest ones (such as Bitcoin and at least the original Ethereum), use Proof-of-Work.

2.8.4. Variants of Proof-of-Stake. DPOS vs. BFT. While Proof-of-Work algorithms are very similar to each other and differ mostly in the hash functions that must be computed for mining new blocks, there are more possibilities for Proof-of-Stake algorithms. They merit a sub-classification of their own.

Essentially, one must answer the following questions about a Proof-of-Stake algorithm:

- Who can produce ("mine") a new block—any full node, or only a member of a (relatively) small subset of validators? (Most PoS systems require new blocks to be generated and signed by one of several designated validators.)
- Do validators guarantee the validity of the blocks by their signatures, or are all full nodes expected to validate all blocks by themselves? (Scalable PoS systems must rely on validator signatures instead of requiring all nodes to validate all blocks of all blockchains.)
- Is there a designated producer for the next blockchain block, known in advance, such that nobody else can produce that block instead?
- Is a newly-created block originally signed by only one validator (its producer), or must it collect a majority of validator signatures from the very beginning?

While there seem to be 2^4 possible classes of PoS algorithms depending on the answers to these questions, the distinction in practice boils down to two major approaches to PoS. In fact, most modern PoS algorithms, designed to be used in scalable multi-chain systems, answer the first two questions in the same fashion: only validators can produce new blocks, and they guarantee block validity without requiring all full nodes to check the validity of all blocks by themselves.

As to the two last questions, their answers turn out to be highly correlated, leaving essentially only two basic options:

- *Delegated Proof-of-Stake (DPOS)*: There is a universally known designated producer for every block; no one else can produce that block; the new block is originally signed only by its producing validator.
- *Byzantine Fault Tolerant (BFT)* PoS algorithms: There is a known subset of validators, any of which can suggest a new block; the choice of the actual next block among several suggested candidates, which must be validated and signed by a majority of validators before being released to the other nodes, is achieved by a version of Byzantine Fault Tolerant consensus protocol.

2.8.5. Comparison of DPOS and BFT PoS. The BFT approach has the advantage that a newly-produced block has *from the very beginning* the

signatures of a majority of validators testifying to its validity. Another advantage is that, if a majority of validators executes the BFT consensus protocol correctly, no forks can appear at all. On the other hand, BFT algorithms tend to be quite convoluted and require more time for the subset of validators to reach consensus. Therefore, blocks cannot be generated too often. This is why we expect the TON Blockchain (which is a BFT project from the perspective of this classification) to produce a block only once every five seconds. In practice, this interval might be decreased to 2–3 seconds (though we do not promise this), but not further, if validators are spread across the globe.

The DPOS algorithm has the advantage of being quite simple and straightforward. It can generate new blocks quite often—say, once every two seconds, or maybe even once every second,[25] because of its reliance on designated block producers known in advance.

However, DPOS requires all nodes—or at least all validators—to validate all blocks received, because a validator producing and signing a new block confirms not only the *relative* validity of this block, but also the validity of the previous block it refers to, and all the blocks further back in the chain (maybe up to the beginning of the period of responsibility of the current subset of validators). There is a predetermined order on the current subset of validators, so that for each block there is a designated producer (i.e., validator expected to generate that block); these designated producers are rotated in a round-robin fashion. In this way, a block is at first signed only by its producing validator; then, when the next block is mined, and its producer chooses to refer to this block and not to one of its predecessors (otherwise its block would lie in a shorter chain, which might lose the "longest fork" competition in the future), the signature of the next block is essentially an additional signature on the previous block as well. In this way, a new block gradually collects the signatures of more validators—say, twenty signatures in the time needed to generate the next twenty blocks. A full node will either need to wait for these twenty signatures, or validate the block by itself, starting from a sufficiently confirmed block (say, twenty blocks back), which might be not so easy.

The obvious disadvantage of the DPOS algorithm is that a new block (and transactions committed into it) achieves the same level of trust ("re-

[25]Some people even claim DPOS block generation times of half a second, which does not seem realistic if validators are scattered across several continents.

cursive reliability" as discussed in **2.6.28**) only after twenty more blocks are mined, compared to the BFT algorithms, which deliver this level of trust (say, twenty signatures) immediately. Another disadvantage is that DPOS uses the "longest fork wins" approach for switching to other forks; this makes forks quite probable if at least some producers fail to produce subsequent blocks after the one we are interested in (or we fail to observe these blocks because of a network partition or a sophisticated attack).

We believe that the BFT approach, while more sophisticated to implement and requiring longer time intervals between blocks than DPOS, is better adapted to "tightly-coupled" (cf. **2.8.14**) multichain systems, because other blockchains can start acting almost immediately after seeing a committed transaction (e.g., generating a message intended for them) in a new block, without waiting for twenty confirmations of validity (i.e., the next twenty blocks), or waiting for the next six blocks to be sure that no forks appear and verifying the new block by themselves (verifying blocks of other blockchains may become prohibitive in a scalable multi-chain system). Thus they can achieve scalability while preserving high reliability and availability (cf. **2.8.12**).

On the other hand, DPOS might be a good choice for a "loosely-coupled" multi-chain system, where fast interaction between blockchains is not required – e.g., if each blockchain ("workchain") represents a separate distributed exchange, and inter-blockchain interaction is limited to rare transfers of tokens from one workchain into another (or, rather, trading one altcoin residing in one workchain for another at a rate approaching $1 : 1$). This is what is actually done in the BitShares project, which uses DPOS quite successfully.

To summarize, while DPOS can *generate* new blocks and *include transactions* into them *faster* (with smaller intervals between blocks), these transactions reach the level of trust required to use them in other blockchains and off-chain applications as "committed" and "immutable" *much more slowly* than in the BFT systems—say, in thirty seconds[26] instead of five. Faster transaction *inclusion* does not mean faster transaction *commitment*. This could become a huge problem if fast inter-blockchain interaction is required. In that case, one must abandon DPOS and opt for BFT PoS instead.

[26]For instance, EOS, one of the best DPOS projects proposed up to this date, promises a 45-second confirmation and inter-blockchain interaction delay (cf. [5], "Transaction Confirmation" and "Latency of Interchain Communication" sections).

2.8.6. Support for Turing-complete code in transactions, i.e., essentially arbitrary smart contracts. Blockchain projects normally collect some *transactions* in their blocks, which alter the blockchain state in a way deemed useful (e.g., transfer some amount of cryptocurrency from one account to another). Some blockchain projects might allow only some specific predefined types of transactions (such as value transfers from one account to another, provided correct signatures are presented). Others might support some limited form of scripting in the transactions. Finally, some blockchains support the execution of arbitrarily complex code in transactions, enabling the system (at least in principle) to support arbitrary applications, provided the performance of the system permits. This is usually associated with "Turing-complete virtual machines and scripting languages" (meaning that any program that can be written in any other computing language may be re-written to be performed inside the blockchain), and "smart contracts" (which are programs residing in the blockchain).

Of course, support for arbitrary smart contracts makes the system truly flexible. On the other hand, this flexibility comes at a cost: the code of these smart contracts must be executed on some virtual machine, and this must be done every time for each transaction in the block when somebody wants to create or validate a block. This slows down the performance of the system compared to the case of a predefined and immutable set of types of simple transactions, which can be optimized by implementing their processing in a language such as C++ (instead of some virtual machine).

Ultimately, support for Turing-complete smart contracts seems to be desirable in any general-purpose blockchain project; otherwise, the designers of the blockchain project must decide in advance which applications their blockchain will be used for. In fact, the lack of support for smart contracts in the Bitcoin blockchain was the principal reason why a new blockchain project, Ethereum, had to be created.

In a (heterogeneous; cf. **2.8.8**) multi-chain system, one might have "the best of both worlds" by supporting Turing-complete smart contracts in some blockchains (i.e., workchains), and a small predefined set of highly-optimized transactions in others.

2.8.7. Classification of multichain systems. So far, the classification was valid both for single-chain and multi-chain systems. However, multichain systems admit several more classification criteria, reflecting the relationship between the different blockchains in the system. We now discuss

these criteria.

2.8.8. Blockchain types: homogeneous and heterogeneous systems. In a multi-chain system, all blockchains may be essentially of the same type and have the same rules (i.e., use the same format of transactions, the same virtual machine for executing smart-contract code, share the same cryptocurrency, and so on), and this similarity is explicitly exploited, but with different data in each blockchain. In this case, we say that the system is *homogeneous*. Otherwise, different blockchains (which will usually be called *workchains* in this case) can have different "rules". Then we say that the system is *heterogeneous*.

2.8.9. Mixed heterogeneous-homogeneous systems. Sometimes we have a mixed system, where there are several sets of types or rules for blockchains, but many blockchains with the same rules are present, and this fact is explicitly exploited. Then it is a mixed *heterogeneous-homogeneous system*. To our knowledge, the TON Blockchain is the only example of such a system.

2.8.10. Heterogeneous systems with several workchains having the same rules, or *confederations*. In some cases, several blockchains (workchains) with the same rules can be present in a heterogeneous system, but the interaction between them is the same as between blockchains with different rules (i.e., their similarity is not exploited explicitly). Even if they appear to use "the same" cryptocurrency, they in fact use different "altcoins" (independent incarnations of the cryptocurrency). Sometimes one can even have certain mechanisms to convert these altcoins at a rate near to $1 : 1$. However, this does not make the system homogeneous in our view; it remains heterogeneous. We say that such a heterogeneous collection of workchains with the same rules is a *confederation*.

While making a heterogeneous system that allows one to create several workchains with the same rules (i.e., a confederation) may seem a cheap way of building a scalable system, this approach has a lot of drawbacks, too. Essentially, if someone hosts a large project in many workchains with the same rules, she does not obtain a large project, but rather a lot of small instances of this project. This is like having a chat application (or a game) that allows having at most 50 members in any chat (or game) room, but "scales" by creating new rooms to accommodate more users when necessary. As a result, a lot of users can participate in the chats or in the game, but

can we say that such a system is truly scalable?

2.8.11. Presence of a masterchain, external or internal. Sometimes, a multi-chain project has a distinguished "masterchain" (sometimes called "control blockchain"), which is used, for example, to store the overall configuration of the system (the set of all active blockchains, or rather workchains), the current set of validators (for a Proof-of-Stake system), and so on. Sometimes other blockchains are "bound" to the masterchain, for example by committing the hashes of their latest blocks into it (this is something the TON Blockchain does, too).

In some cases, the masterchain is *external*, meaning that it is not a part of the project, but some other pre-existing blockchain, originally completely unrelated to its use by the new project and agnostic of it. For example, one can try to use the Ethereum blockchain as a masterchain for an external project, and publish special smart contracts into the Ethereum blockchain for this purpose (e.g., for electing and punishing validators).

2.8.12. Sharding support. Some blockchain projects (or systems) have native support for *sharding*, meaning that several (necessarily homogeneous; cf. **2.8.8**) blockchains are thought of as *shards* of a single (from a high-level perspective) virtual blockchain. For example, one can create 256 shard blockchains ("shardchains") with the same rules, and keep the state of an account in exactly one shard selected depending on the first byte of its *account_id*.

Sharding is a natural approach to scaling blockchain systems, because, if it is properly implemented, users and smart contracts in the system need not be aware of the existence of sharding at all. In fact, one often wants to add sharding to an existing single-chain project (such as Ethereum) when the load becomes too high.

An alternative approach to scaling would be to use a "confederation" of heterogeneous workchains as described in **2.8.10**, allowing each user to keep her account in one or several workchains of her choice, and transfer funds from her account in one workchain to another workchain when necessary, essentially performing a $1:1$ altcoin exchange operation. The drawbacks of this approach have already been discussed in **2.8.10**.

However, sharding is not so easy to implement in a fast and reliable fashion, because it implies a lot of messages between different shardchains. For example, if accounts are evenly distributed between N shards, and the only transactions are simple fund transfers from one account to another, then only

a small fraction $(1/N)$ of all transactions will be performed within a single blockchain; almost all $(1 - 1/N)$ transactions will involve two blockchains, requiring inter-blockchain communication. If we want these transactions to be fast, we need a fast system for transferring messages between shardchains. In other words, the blockchain project needs to be "tightly-coupled" in the sense described in **2.8.14**.

2.8.13. Dynamic and static sharding. Sharding might be *dynamic* (if additional shards are automatically created when necessary) or *static* (when there is a predefined number of shards, which is changeable only through a hard fork at best). Most sharding proposals are static; the TON Blockchain uses dynamic sharding (cf. **2.7**).

2.8.14. Interaction between blockchains: loosely-coupled and tightly-coupled systems. Multi-blockchain projects can be classified according to the supported level of interaction between the constituent blockchains.

The least level of support is the absence of any interaction between different blockchains whatsoever. We do not consider this case here, because we would rather say that these blockchains are not parts of one blockchain system, but just separate instances of the same blockchain protocol.

The next level of support is the absence of any specific support for messaging between blockchains, making interaction possible in principle, but awkward. We call such systems "loosely-coupled"; in them one must send messages and transfer value between blockchains as if they had been blockchains belonging to completely separate blockchain projects (e.g., Bitcoin and Ethereum; imagine two parties want to exchange some Bitcoins, kept in the Bitcoin blockchain, into Ethers, kept in the Ethereum blockchain). In other words, one must include the outbound message (or its generating transaction) in a block of the source blockchain. Then she (or some other party) must wait for enough confirmations (e.g., a given number of subsequent blocks) to consider the originating transaction to be "committed" and "immutable", so as to be able to perform external actions based on its existence. Only then may a transaction relaying the message into the target blockchain (perhaps along with a reference and a Merkle proof of existence for the originating transaction) be committed.

If one does not wait long enough before transferring the message, or if a fork happens anyway for some other reason, the joined state of the two blockchains turns out to be inconsistent: a message is delivered into the

second blockchain that has never been generated in (the ultimately chosen fork of) the first blockchain.

Sometimes partial support for messaging is added, by standardizing the format of messages and the location of input and output message queues in the blocks of all workchains (this is especially useful in heterogeneous systems). While this facilitates messaging to a certain extent, it is conceptually not too different from the previous case, so such systems are still "loosely-coupled".

By contrast, "tightly-coupled" systems include special mechanisms to provide fast messaging between all blockchains. The desired behavior is to be able to deliver a message into another workchain immediately after it has been generated in a block of the originating blockchain. On the other hand, "tightly-coupled" systems are also expected to maintain overall consistency in the case of forks. While these two requirements appear to be contradictory at first glance, we believe that the mechanisms used by the TON Blockchain (the inclusion of shardchain block hashes into masterchain blocks; the use of "vertical" blockchains for fixing invalid blocks, cf. **2.1.17**; hypercube routing, cf. **2.4.19**; Instant Hypercube Routing, cf. **2.4.20**) enable it to be a "tightly-coupled" system, perhaps the only one so far.

Of course, building a "loosely-coupled" system is much simpler; however, fast and efficient sharding (cf. **2.8.12**) requires the system to be "tightly-coupled".

2.8.15. Simplified classification. Generations of blockchain projects. The classification we have suggested so far splits all blockchain projects into a large number of classes. However, the classification criteria we use happen to be quite correlated in practice. This enables us to suggest a simplified "generational" approach to the classification of blockchain projects, as a very rough approximation of reality, with some examples. Projects that have not been implemented and deployed yet are shown in *italics*; the most important characteristics of a generation are shown in **bold**.

- First generation: Single-chain, **PoW**, no support for smart contracts. Examples: Bitcoin (2009) and a lot of otherwise uninteresting imitators (Litecoin, Monero, ...).
- Second generation: Single-chain, PoW, **smart-contract support**. Example: Ethereum (2013; deployed in 2015), at least in its original form.

- Third generation: Single-chain, **PoS**, smart-contract support. Example: *future Ethereum* (2018 or later).
- Alternative third (3′) generation: **Multi-chain**, PoS, no support for smart contracts, loosely-coupled. Example: Bitshares (2013–2014; uses DPOS).
- Fourth generation: **Multi-chain, PoS, smart-contract support**, loosely-coupled. Examples: *EOS* (2017; uses DPOS), *PolkaDot* (2016; uses BFT).
- Fifth generation: Multi-chain, PoS with BFT, smart-contract support, **tightly-coupled, with sharding**. Examples: *TON* (2017).

While not all blockchain projects fall precisely into one of these categories, most of them do.

2.8.16. Complications of changing the "genome" of a blockchain project. The above classification defines the "genome" of a blockchain project. This genome is quite "rigid": it is almost impossible to change it once the project is deployed and is used by a lot of people. One would need a series of hard forks (which would require the approval of the majority of the community), and even then the changes would need to be very conservative in order to preserve backward compatibility (e.g., changing the semantics of the virtual machine might break existing smart contracts). An alternative would be to create new "sidechains" with their different rules, and bind them somehow to the blockchain (or the blockchains) of the original project. One might use the blockchain of the existing single-blockchain project as an external masterchain for an essentially new and separate project.[27]

Our conclusion is that the genome of a project is very hard to change once it has been deployed. Even starting with PoW and planning to replace it with PoS in the future is quite complicated.[28] Adding shards to a project originally designed without support for them seems almost impossible.[29] In

[27]For example, the Plasma project plans to use the Ethereum blockchain as its (external) masterchain; it does not interact much with Ethereum otherwise, and it could have been suggested and implemented by a team unrelated to the Ethereum project.

[28]As of 2017, Ethereum is still struggling to transition from PoW to a combined PoW+PoS system; we hope it will become a truly PoS system someday.

[29]There are sharding proposals for Ethereum dating back to 2015; it is unclear how they might be implemented and deployed without disrupting Ethereum or creating an essentially independent parallel project.

fact, adding support for smart contracts into a project (namely, Bitcoin) originally designed without support for such features has been deemed impossible (or at least undesirable by the majority of the Bitcoin community) and eventually led to the creation of a new blockchain project, Ethereum.

2.8.17. Genome of the TON Blockchain. Therefore, if one wants to build a scalable blockchain system, one must choose its genome carefully from the very beginning. If the system is meant to support some additional specific functionality in the future not known at the time of its deployment, it should support "heterogeneous" workchains (having potentially different rules) from the start. For the system to be truly scalable, it must support sharding from the very beginning; sharding makes sense only if the system is "tightly-coupled" (cf. **2.8.14**), so this in turn implies the existence of a masterchain, a fast system of inter-blockchain messaging, usage of BFT PoS, and so on.

When one takes into account all these implications, most of the design choices made for the TON Blockchain project appear natural, and almost the only ones possible.

2.9 Comparison to Other Blockchain Projects

We conclude our brief discussion of the TON Blockchain and its most important and unique features by trying to find a place for it on a map containing existing and proposed blockchain projects. We use the classification criteria described in **2.8** to discuss different blockchain projects in a uniform way and construct such a "map of blockchain projects". We represent this map as Table 1, and then briefly discuss a few projects separately to point out their peculiarities that may not fit into the general scheme.

2.9.1. Bitcoin [12]; `https://bitcoin.org/`. *Bitcoin* (2009) is the first and the most famous blockchain project. It is a typical *first-generation* blockchain project: it is single-chain, it uses Proof-of-Work with a "longest-fork-wins" fork selection algorithm, and it does not have a Turing-complete scripting language (however, simple scripts without loops are supported). The Bitcoin blockchain has no notion of an account; it uses the UTXO (Unspent Transaction Output) model instead.

2.9.2. Ethereum [2]; `https://ethereum.org/`. *Ethereum* (2015) is the first blockchain with support for Turing-complete smart contracts. As such,

Project	Year	G.	Cons.	Sm.	Ch.	R.	Sh.	Int.
Bitcoin	2009	1	PoW	no	1			
Ethereum	2013, 2015	2	PoW	yes	1			
NXT	2014	2+	PoS	no	1			
Tezos	2017, ?	2+	PoS	yes	1			
Casper	2015, (2017)	3	PoW/PoS	yes	1			
BitShares	2013, 2014	3′	DPoS	no	m	ht.	no	L
EOS	2016, (2018)	4	DPoS	yes	m	ht.	no	L
PolkaDot	2016, (2019)	4	PoS BFT	yes	m	ht.	no	L
Cosmos	2017, ?	4	PoS BFT	yes	m	ht.	no	L
TON	2017, (2018)	5	PoS BFT	yes	m	mix	dyn.	T

Table 1: A summary of some notable blockchain projects. The columns are: *Project* – project name; *Year* – year announced and year deployed; *G.* – generation (cf. **2.8.15**); *Cons.* – consensus algorithm (cf. **2.8.3** and **2.8.4**); *Sm.* – support for arbitrary code (smart contracts; cf. **2.8.6**); *Ch.* – single/multiple blockchain system (cf. **2.8.2**); *R.* – heterogeneous/homogeneous multichain systems (cf. **2.8.8**); *Sh.* – sharding support (cf. **2.8.12**); *Int.* – interaction between blockchains, (L)oose or (T)ight (cf. **2.8.14**).

it is a typical *second-generation* project, and the most popular among them. It uses Proof-of-Work on a single blockchain, but has smart contracts and accounts.

2.9.3. NXT; `https://nxtplatform.org/`. *NXT* (2014) is the first PoS-based blockchain and currency. It is still single-chain, and has no smart contract support.

2.9.4. Tezos; `https://www.tezos.com/`. *Tezos* (2018 or later) is a proposed PoS-based single-blockchain project. We mention it here because of its unique feature: its block interpretation function *ev_block* (cf. **2.2.6**) is not fixed, but is determined by an OCaml module, which can be upgraded by committing a new version into the blockchain (and collecting some votes for the proposed change). In this way, one will be able to create custom single-chain projects by first deploying a "vanilla" Tezos blockchain, and then gradually changing the block interpretation function in the desired direction, without any need for hard forks.

This idea, while intriguing, has the obvious drawback that it forbids any optimized implementations in other languages like C++, so a Tezos-based blockchain is destined to have lower performance. We think that a similar result might have been obtained by publishing a formal *specification* of the proposed block interpretation function *ev_trans*, without fixing a particular

implementation.

2.9.5. Casper.[30] *Casper* is an upcoming PoS algorithm for Ethereum; its gradual deployment in 2017 (or 2018), if successful, will change Ethereum into a single-chain PoS or mixed PoW+PoS system with smart contract support, transforming Ethereum into a *third-generation* project.

2.9.6. BitShares [8]; `https://bitshares.org`. *BitShares* (2014) is a platform for distributed blockchain-based exchanges. It is a heterogeneous multi-blockchain DPoS system without smart contracts; it achieves its high performance by allowing only a small set of predefined specialized transaction types, which can be efficiently implemented in C++, assuming the blockchain state fits into memory. It is also the first blockchain project to use Delegated Proof-of-Stake (DPoS), demonstrating its viability at least for some specialized purposes.

2.9.7. EOS [5]; `https://eos.io`. *EOS* (2018 or later) is a proposed heterogeneous multi-blockchain DPoS system *with* smart contract support and with some minimal support for messaging (still loosely-coupled in the sense described in **2.8.14**). It is an attempt by the same team that has previously successfully created the BitShares and SteemIt projects, demonstrating the strong points of the DPoS consensus algorithm. Scalability will be achieved by creating specialized workchains for projects that need it (e.g., a distributed exchange might use a workchain supporting a special set of optimized transactions, similarly to what BitShares did) and by creating multiple workchains with the same rules (*confederations* in the sense described in **2.8.10**). The drawbacks and limitations of this approach to scalability have been discussed in *loc. cit.* Cf. also **2.8.5**, **2.8.12**, and **2.8.14** for a more detailed discussion of DPoS, sharding, interaction between workchains and their implications for the scalability of a blockchain system.

At the same time, even if one will not be able to "create a Facebook inside a blockchain" (cf. **2.9.13**), EOS or otherwise, we think that EOS might become a convenient platform for some highly-specialized weakly interacting distributed applications, similar to BitShares (decentralized exchange) and SteemIt (decentralized blog platform).

2.9.8. PolkaDot [17]; `https://polkadot.io/`. *PolkaDot* (2019 or later) is one of the best thought-out and most detailed proposed multichain Proof-

[30] `https://blog.ethereum.org/2015/08/01/introducing-casper-friendly-ghost/`

of-Stake projects; its development is led by one of the Ethereum co-founders. This project is one of the closest projects to the TON Blockchain on our map. (In fact, we are indebted for our terminology for "fishermen" and "nominators" to the PolkaDot project.)

PolkaDot is a heterogeneous loosely-coupled multichain Proof-of-Stake project, with Byzantine Fault Tolerant (BFT) consensus for generation of new blocks and a masterchain (which might be external—e.g., the Ethereum blockchain). It also uses hypercube routing, somewhat like (the slow version of) TON's as described in **2.4.19**.

Its unique feature is its ability to create not only *public*, but also *private* blockchains. These private blockchains would also be able to interact with other public blockchains, PolkaDot or otherwise.

As such, PolkaDot might become a platform for large-scale *private* blockchains, which might be used, for example, by bank consortiums to quickly transfer funds to each other, or for any other uses a large corporation might have for private blockchain technology.

However, PolkaDot has no sharding support and is not tightly-coupled. This somewhat hampers its scalability, which is similar to that of EOS. (Perhaps a bit better, because PolkaDot uses BFT PoS instead of DPoS.)

2.9.9. Universa; `https://universa.io`. The only reason we mention this unusual blockchain project here is because it is the only project so far to make in passing an explicit reference to something similar to our Infinite Sharding Paradigm (cf. **2.1.2**). Its other peculiarity is that it bypasses all complications related to Byzantine Fault Tolerance by promising that only trusted and licensed partners of the project will be admitted as validators, hence they will never commit invalid blocks. This is an interesting decision; however, it essentially makes a blockchain project deliberately *centralized*, something blockchain projects usually want to avoid (why does one need a blockchain at all to work in a trusted centralized environment?).

2.9.10. Plasma; `https://plasma.io`). *Plasma* (2019?) is an unconventional blockchain project from another co-founder of Ethereum. It is supposed to mitigate some limitations of Ethereum without introducing sharding. In essence, it is a separate project from Ethereum, introducing a hierarchy of (heterogeneous) workchains, bound to the Ethereum blockchain (to be used as an external masterchain) at the top level. Funds can be transferred from any blockchain up in the hierarchy (starting from the Ethereum blockchain as the root), along with a description of a job to be done. Then

the necessary computations are done in the child workchain (possibly requiring forwarding of parts of the original job further down the tree), their results are passed up, and a reward is collected. The problem of achieving consistency and validating these workchains is circumvented by a (payment channel-inspired) mechanism allowing users to unilaterally withdraw their funds from a misbehaving workchain to its parent workchain (albeit slowly), and re-allocate their funds and their jobs to another workchain.

In this way, Plasma might become a platform for distributed computations bound to the Ethereum blockchain, something like a "mathematical co-processor". However, this does not seem like a way to achieve true general-purpose scalability.

2.9.11. Specialized blockchain projects. There are also some specialized blockchain projects, such as FileCoin (a system that incentivizes users to offer their disk space for storing the files of other users who are willing to pay for it), Golem (a blockchain-based platform for renting and lending computing power for specialized applications such as 3D-rendering) or SONM (another similar computing power-lending project). Such projects do not introduce anything conceptually new on the level of blockchain organization; rather, they are particular blockchain applications, which could be implemented by smart contracts running in a general-purpose blockchain, provided it can deliver the required performance. As such, projects of this kind are likely to use one of the existing or planned blockchain projects as their base, such as EOS, PolkaDot or TON. If a project needs "true" scalability (based on sharding), it would better use TON; if it is content to work in a "confederated" context by defining a family of workchains of its own, explicitly optimized for its purpose, it might opt for EOS or PolkaDot.

2.9.12. The TON Blockchain. The TON (Telegram Open Network) Blockchain (planned 2018) is the project we are describing in this document. It is designed to be the first fifth-generation blockchain project—that is, a BFT PoS-multichain project, mixed homogeneous/heterogeneous, with support for (shardable) custom workchains, with native sharding support, and tightly-coupled (in particular, capable of forwarding messages between shards almost instantly while preserving a consistent state of all shardchains). As such, it will be a truly scalable general-purpose blockchain project, capable of accommodating essentially any applications that can be implemented in a blockchain at all. When augmented by the other components of the TON Project (cf. **1**), its possibilities expand even further.

2.9.13. Is it possible to "upload Facebook into a blockchain"? Sometimes people claim that it will be possible to implement a social network on the scale of Facebook as a distributed application residing in a blockchain. Usually a favorite blockchain project is cited as a possible "host" for such an application.

We cannot say that this is a technical impossibility. Of course, one needs a tightly-coupled blockchain project with true sharding (i.e., TON) in order for such a large application not to work too slowly (e.g., deliver messages and updates from users residing in one shardchain to their friends residing in another shardchain with reasonable delays). However, we think that this is not needed and will never be done, because the price would be prohibitive.

Let us consider "uploading Facebook into a blockchain" as a thought experiment; any other project of similar scale might serve as an example as well. Once Facebook is uploaded into a blockchain, all operations currently done by Facebook's servers will be serialized as transactions in certain blockchains (e.g., TON's shardchains), and will be performed by all validators of these blockchains. Each operation will have to be performed, say, at least twenty times, if we expect every block to collect at least twenty validator signatures (immediately or eventually, as in DPOS systems). Similarly, all data kept by Facebook's servers on their disks will be kept on the disks of all validators for the corresponding shardchain (i.e., in at least twenty copies).

Because the validators are essentially the same servers (or perhaps clusters of servers, but this does not affect the validity of this argument) as those currently used by Facebook, we see that the total hardware expenses associated with running Facebook in a blockchain are at least twenty times higher than if it were implemented in the conventional way.

In fact, the expenses would be much higher still, because the blockchain's virtual machine is slower than the "bare CPU" running optimized compiled code, and its storage is not optimized for Facebook-specific problems. One might partially mitigate this problem by crafting a specific workchain with some special transactions adapted for Facebook; this is the approach of BitShares and EOS to achieving high performance, available in the TON Blockchain as well. However, the general blockchain design would still impose some additional restrictions by itself, such as the necessity to register all operations as transactions in a block, to organize these transactions in a Merkle tree, to compute and check their Merkle hashes, to propagate this block further, and so on.

Therefore, a conservative estimate is that one would need 100 times more

servers of the same performance as those used by Facebook now in order to validate a blockchain project hosting a social network of that scale. Somebody will have to pay for these servers, either the company owning the distributed application (imagine seeing 700 ads on each Facebook page instead of 7) or its users. Either way, this does not seem economically viable.

We believe that *it is not true that everything should be uploaded into the blockchain*. For example, it is not necessary to keep user photographs in the blockchain; registering the hashes of these photographs in the blockchain and keeping the photographs in a distributed off-chain storage (such as FileCoin or TON Storage) would be a better idea. This is the reason why TON is not just a blockchain project, but a collection of several components (TON P2P Network, TON Storage, TON Services) centered around the TON Blockchain as outlined in Chapters **1** and **4**.

3 TON Networking

Any blockchain project requires not only a specification of block format and blockchain validation rules, but also a network protocol used to propagate new blocks, send and collect transaction candidates and so on. In other words, a specialized peer-to-peer network must be set up by every blockchain project. This network must be peer-to-peer, because blockchain projects are normally expected to be decentralized, so one cannot rely on a centralized group of servers and use conventional client-server architecture, as, for instance, classical online banking applications do. Even light clients (e.g., light cryptocurrency wallet smartphone applications), which must connect to full nodes in a client-server–like fashion, are actually free to connect to another full node if their previous peer goes down, provided the protocol used to connect to full nodes is standardized enough.

While the networking demands of single-blockchain projects, such as Bitcoin or Ethereum, can be met quite easily (one essentially needs to construct a "random" peer-to-peer overlay network, and propagate all new blocks and transaction candidates by a gossip protocol), multi-blockchain projects, such as the TON Blockchain, are much more demanding (e.g., one must be able to subscribe to updates of only some shardchains, not necessarily all of them). Therefore, the networking part of the TON Blockchain and the TON Project as a whole merits at least a brief discussion.

On the other hand, once the more sophisticated network protocols needed to support the TON Blockchain are in place, it turns out that they can easily be used for purposes not necessarily related to the immediate demands of the TON Blockchain, thus providing more possibilities and flexibility for creating new services in the TON ecosystem.

3.1 Abstract Datagram Network Layer

The cornerstone in building the TON networking protocols is the *(TON) Abstract (Datagram) Network Layer*. It enables all nodes to assume certain "network identities", represented by 256-bit "abstract network addresses", and communicate (send datagrams to each other, as a first step) using only these 256-bit network addresses to identify the sender and the receiver. In particular, one does not need to worry about IPv4 or IPv6 addresses, UDP port numbers, and the like; they are hidden by the Abstract Network Layer.

3.1.1. Abstract network addresses. An *abstract network address*, or an *abstract address*, or just *address* for short, is a 256-bit integer, essentially equal to a 256-bit ECC public key. This public key can be generated arbitrarily, thus creating as many different network identities as the node likes. However, one must know the corresponding *private* key in order to receive (and decrypt) messages intended for such an address.

In fact, the address is *not* the public key itself; instead, it is a 256-bit hash (Hash = sha256) of a serialized TL-object (cf. **2.2.5**) that can describe several types of public keys and addresses depending on its constructor (first four bytes). In the simplest case, this serialized TL-object consists just of a 4-byte magic number and a 256-bit elliptic curve cryptography (ECC) public key; in this case, the address will equal the hash of this 36-byte structure. One might use, however, 2048-bit RSA keys, or any other scheme of public-key cryptography instead.

When a node learns another node's abstract address, it must also receive its "preimage" (i.e., the serialized TL-object, the hash of which equals that abstract address) or else it will not be able to encrypt and send datagrams to that address.

3.1.2. Lower-level networks. UDP implementation. From the perspective of almost all TON Networking components, the only thing that exists is a network (the Abstract Datagram Networking Layer) able to (unreliably) send datagrams from one abstract address to another. In principle, the Abstract Datagram Networking Layer (ADNL) can be implemented over different existing network technologies. However, we are going to implement it over UDP in IPv4/IPv6 networks (such as the Internet or intranets), with an optional TCP fallback if UDP is not available.

3.1.3. Simplest case of ADNL over UDP. The simplest case of sending a datagram from a sender's abstract address to any other abstract address (with known preimage) can be implemented as follows.

Suppose that the sender somehow knows the IP-address and the UDP port of the receiver who owns the destination abstract address, and that both the receiver and the sender use abstract addresses derived from 256-bit ECC public keys.

In this case, the sender simply augments the datagram to be sent by its ECC signature (done with its private key) and its source address (or the preimage of the source address, if the receiver is not known to know that

preimage yet). The result is encrypted with the recipient's public key, embedded into a UDP datagram and sent to the known IP and port of the recipient. Because the first 256 bits of the UDP datagram contain the recipient's abstract address, the recipient can identify which private key should be used to decrypt the remainder of the datagram. Only after that is the sender's identity revealed.

3.1.4. Less secure way, with the sender's address in plaintext. Sometimes a less secure scheme is sufficient, when the recipient's and the sender's addresses are kept in plaintext in the UDP datagram; the sender's private key and the recipient's public key are combined together using ECDH (Elliptic Curve Diffie–Hellman) to generate a 256-bit shared secret, which is used afterwards, along with a random 256-bit nonce also included in the unencrypted part, to derive AES keys used for encryption. The integrity may be provided, for instance, by concatenating the hash of the original plaintext data to the plaintext before encryption.

This approach has the advantage that, if more than one datagram is expected to be exchanged between the two addresses, the shared secret can be computed only once and then cached; then slower elliptic curve operations will no longer be required for encrypting or decrypting the next datagrams.

3.1.5. Channels and channel identifiers. In the simplest case, the first 256 bits of a UDP datagram carrying an embedded TON ADNL datagram will be equal to the recipient's address. However, in general they constitute a *channel identifier*. There are different types of channels. Some of them are point-to-point; they are created by two parties who wish to exchange a lot of data in the future and generate a shared secret by exchanging several packets encrypted as described in **3.1.3** or **3.1.4**, by running classical or elliptic curve Diffie–Hellman (if extra security is required), or simply by one party generating a random shared secret and sending it to the other party.

After that, a channel identifier is derived from the shared secret combined with some additional data (such as the sender's and recipient's addresses), for instance by hashing, and that identifier is used as the first 256 bits of UDP datagrams carrying data encrypted with the aid of that shared secret.

3.1.6. Channel as a tunnel identifier. In general, a "channel", or "channel identifier" simply selects a way of processing an inbound UDP datagram, known to the receiver. If the channel is the receiver's abstract address, the processing is done as outlined in **3.1.3** or **3.1.4**; if the channel is an estab-

lished point-to-point channel discussed in **3.1.5**, the processing consists in decrypting the datagram with the aid of the shared secret as explained in *loc. cit.*, and so on.

In particular, a channel identifier can actually select a "tunnel", when the immediate recipient simply forwards the received message to somebody else—the actual recipient or another proxy. Some encryption or decryption steps (reminiscent of "onion routing" [6] or even "garlic routing"[31]) might be done along the way, and another channel identifier might be used for re-encrypted forwarded packets (for example, a peer-to-peer channel could be employed to forward the packet to the next recipient on the path).

In this way, some support for "tunneling" and "proxying"—somewhat similar to that provided by the TOR or I^2P projects—can be added on the level of the TON Abstract Datagram Network Layer, without affecting the functionality of all higher-level TON network protocols, which would be agnostic of such an addition. This opportunity is exploited by the *TON Proxy* service (cf. **4.1.11**).

3.1.7. Zero channel and the bootstrap problem. Normally, a TON ADNL node will have some "neighbor table", containing information about other known nodes, such as their abstract addresses and their preimages (i.e., public keys) and their IP addresses and UDP ports. Then it will gradually extend this table by using information learned from these known nodes as answers to special queries, and sometimes prune obsolete records.

However, when a TON ADNL node just starts up, it may happen that it does not know any other node, and can learn only the IP address and UDP port of a node, but not its abstract address. This happens, for example, if a light client is not able to access any of the previously cached nodes and any nodes hardcoded into the software, and must ask the user to enter an IP address or a DNS domain of a node, to be resolved through DNS.

In this case, the node will send packets to a special "zero channel" of the node in question. This does not require knowledge of the recipient's public key (but the message should still contain the sender's identity and signature), so the message is transferred without encryption. It should be normally used only to obtain an identity (maybe a one-time identity created especially for this purpose) of the receiver, and then to start communicating in a safer way.

Once at least one node is known, it is easy to populate the "neighbor table" and "routing table" by more entries, learning them from answers to

[31] `https://geti2p.net/en/docs/how/garlic-routing`

special queries sent to the already known nodes.

Not all nodes are required to process datagrams sent to the zero channel, but those used to bootstrap light clients should support this feature.

3.1.8. TCP-like stream protocol over ADNL. The ADNL, being an unreliable (small-size) datagram protocol based on 256-bit abstract addresses, can be used as a base for more sophisticated network protocols. One can build, for example, a TCP-like stream protocol, using ADNL as an abstract replacement for IP. However, most components of the TON Project do not need such a stream protocol.

3.1.9. RLDP, or Reliable Large Datagram Protocol over ADNL. A reliable arbitrary-size datagram protocol built upon the ADNL, called RLDP, is used instead of a TCP-like protocol. This reliable datagram protocol can be employed, for instance, to send RPC queries to remote hosts and receive answers from them (cf. **4.1.5**).

3.2 TON DHT: Kademlia-like Distributed Hash Table

The *TON Distributed Hash Table (DHT)* plays a crucial role in the networking part of the TON Project, being used to locate other nodes in the network. For example, a client wanting to commit a transaction into a shardchain might want to find a validator or a collator of that shardchain, or at least some node that might relay the client's transaction to a collator. This can be done by looking up a special key in the TON DHT. Another important application of the TON DHT is that it can be used to quickly populate a new node's neighbor table (cf. **3.1.7**), simply by looking up a random key, or the new node's address. If a node uses proxying and tunneling for its inbound datagrams, it publishes the tunnel identifier and its entry point (e.g., IP address and UDP port) in the TON DHT; then all nodes wishing to send datagrams to that node will obtain this contact information from the DHT first.

The TON DHT is a member of the family of *Kademlia-like distributed hash tables* [10].

3.2.1. Keys of the TON DHT. The *keys* of the TON DHT are simply 256-bit integers. In most cases, they are computed as SHA256 of a TL-serialized object (cf. **2.2.5**), called *preimage* of the key, or *key description*. In some cases, the abstract addresses of the TON Network nodes (cf. **3.1.1**) can also

be used as keys of the TON DHT, because they are also 256-bit, and they are also hashes of TL-serialized objects. For example, if a node is not afraid of publishing its IP address, it can be found by anybody who knows its abstract address by simply looking up that address as a key in the DHT.

3.2.2. Values of the DHT. The *values* assigned to these 256-bit keys are essentially arbitrary byte strings of limited length. The interpretation of such byte strings is determined by the preimage of the corresponding key; it is usually known both by the node that looks up the key, and by the node that stores the key.

3.2.3. Nodes of the DHT. Semi-permanent network identities. The key-value mapping of the TON DHT is kept on the *nodes* of the DHT—essentially, all members of the TON Network. To this end, any node of the TON Network (perhaps with the exception of some very light nodes), apart from any number of ephemeral and permanent abstract addresses described in **3.1.1**, has at least one "semi-permanent address", which identifies it as a member of the TON DHT. This *semi-permanent* or *DHT address* should not to be changed too often, otherwise other nodes would be unable to locate the keys they are looking for. If a node does not want to reveal its "true" identity, it generates a separate abstract address to be used only for the purpose of participating in the DHT. However, this abstract address must be public, because it will be associated with the node's IP address and port.

3.2.4. Kademlia distance. Now we have both 256-bit keys and 256-bit (semi-permanent) node addresses. We introduce the so-called *XOR distance* or *Kademlia distance* d_K on the set of 256-bit sequences, given by

$$d_K(x,y) := (x \oplus y) \quad \text{interpreted as an unsigned 256-bit integer} \tag{25}$$

Here $x \oplus y$ denotes the bitwise eXclusive OR (XOR) of two bit sequences of the same length.

The Kademlia distance introduces a metric on the set $\mathbf{2}^{256}$ of all 256-bit sequences. In particular, we have $d_K(x,y) = 0$ if and only if $x = y$, $d_K(x,y) = d_K(y,x)$, and $d_K(x,z) \le d_K(x,y) + d_K(y,z)$. Another important property is that *there is only one point at any given distance from* x: $d_K(x,y) = d_K(x,y')$ implies $y = y'$.

3.2.5. Kademlia-like DHTs and the TON DHT. We say that a distributed hash table (DHT) with 256-bit keys and 256-bit node addresses is a

Kademlia-like DHT if it is expected to keep the value of key K on s Kademlia-nearest nodes to K (i.e., the s nodes with smallest Kademlia distance from their addresses to K.)

Here s is a small parameter, say, $s = 7$, needed to improve reliability of the DHT (if we would keep the key only on one node, the nearest one to K, the value of that key would be lost if that only node goes offline).

The TON DHT is a Kademlia-like DHT, according to this definition. It is implemented over the ADNL protocol described in **3.1**.

3.2.6. Kademlia routing table. Any node participating in a Kademlia-like DHT usually maintains a *Kademlia routing table*. In the case of TON DHT, it consists of $n = 256$ buckets, numbered from 0 to $n - 1$. The i-th bucket will contain information about some known nodes (a fixed number t of "best" nodes, and maybe some extra candidates) that lie at a Kademlia distance from 2^i to $2^{i+1} - 1$ from the node's address a.[32] This information includes their (semi-permanent) addresses, IP addresses and UDP ports, and some availability information such as the time and the delay of the last ping.

When a Kademlia node learns about any other Kademlia node as a result of some query, it includes it into a suitable bucket of its routing table, first as a candidate. Then, if some of the "best" nodes in that bucket fail (e.g., do not respond to ping queries for a long time), they can be replaced by some of the candidates. In this way the Kademlia routing table stays populated.

New nodes from the Kademlia routing table are also included in the ADNL neighbor table described in **3.1.7**. If a "best" node from a bucket of the Kademlia routing table is used often, a channel in the sense described in **3.1.5** can be established to facilitate the encryption of datagrams.

A special feature of the TON DHT is that it tries to select nodes with the smallest round-trip delays as the "best" nodes for the buckets of the Kademlia routing table.

3.2.7. (Kademlia network queries.) A Kademlia node usually supports the following network queries:

- Ping – Checks node availability.

[32] If there are sufficiently many nodes in a bucket, it can be subdivided further into, say, eight sub-buckets depending on the top four bits of the Kademlia distance. This would speed up DHT lookups.

- Store($key, value$) – Asks the node to keep $value$ as a value for key key. For TON DHT, the Store queries are slightly more complicated (cf. **3.2.9**).
- Find_Node(key, l) – Asks the node to return l Kademlia-nearest known nodes (from its Kademlia routing table) to key.
- Find_Value(key, l) – The same as above, but if the node knows the value corresponding to key key, it just returns that value.

When any node wants to look up the value of a key K, it first creates a set S of s' nodes (for some small value of s', say, $s' = 5$), nearest to K with respect to the Kademlia distance among all known nodes (i.e., they are taken from the Kademlia routing table). Then a Find_Value query is sent to each of them, and nodes mentioned in their answers are included in S. Then the s' nodes from S, nearest to K, are also sent a Find_Value query if this hasn't been done before, and the process continues until the value is found or the set S stops growing. This is a sort of "beam search" of the node nearest to K with respect to Kademlia distance.

If the value of some key K is to be set, the same procedure is run for $s' \geq s$, with Find_Node queries instead of Find_Value, to find s nearest nodes to K. Afterwards, Store queries are sent to all of them.

There are some less important details in the implementation of a Kademlia-like DHT (for example, any node should look up s nearest nodes to itself, say, once every hour, and re-publish all stored keys to them by means of Store queries). We will ignore them for the time being.

3.2.8. Booting a Kademlia node. When a Kademlia node goes online, it first populates its Kademlia routing table by looking up its own address. During this process, it identifies the s nearest nodes to itself. It can download from them all $(key, value)$ pairs known to them to populate its part of the DHT.

3.2.9. Storing values in TON DHT. Storing values in TON DHT is slightly different from a general Kademlia-like DHT. When someone wishes to store a value, she must provide not only the key K itself to the Store query, but also its *preimage*—i.e., a TL-serialized string (with one of several predefined TL-constructors at the beginning) containing a "description" of the key. This key description is later kept by the node, along with the key and the value.

The key description describes the "type" of the object being stored, its "owner", and its "update rules" in case of future updates. The owner is usually identified by a public key included in the key description. If it is included, normally only updates signed by the corresponding private key will be accepted. The "type" of the stored object is normally just a byte string. However, in some cases it can be more sophisticated—for example, an input tunnel description (cf. **3.1.6**), or a collection of node addresses.

The "update rules" can also be different. In some cases, they simply permit replacing the old value with the new value, provided the new value is signed by the owner (the signature must be kept as part of the value, to be checked later by any other nodes after they obtain the value of this key). In other cases, the old value somehow affects the new value. For example, it can contain a sequence number, and the old value is overwritten only if the new sequence number is larger (to prevent replay attacks).

3.2.10. Distributed "torrent trackers" and "network interest groups" in TON DHT. Yet another interesting case is when the value contains a list of nodes—perhaps with their IP addresses and ports, or just with their abstract addresses—and the "update rule" consists in including the requester in this list, provided she can confirm her identity.

This mechanism might be used to create a distributed "torrent tracker", where all nodes interested in a certain "torrent" (i.e., a certain file) can find other nodes that are interested in the same torrent, or already have a copy.

TON Storage (cf. **4.1.8**) uses this technology to find the nodes that have a copy of a required file (e.g., a snapshot of the state of a shardchain, or an old block). However, its more important use is to create "overlay multicast subnetworks" and "network interest groups" (cf. **3.3**). The idea is that only some nodes are interested in the updates of a specific shardchain. If the number of shardchains becomes very large, finding even one node interested in the same shard may become complicated. This "distributed torrent tracker" provides a convenient way to find some of these nodes. Another option would be to request them from a validator, but this would not be a scalable approach, and validators might choose not to respond to such queries coming from arbitrary unknown nodes.

3.2.11. Fall-back keys. Most of the "key types" described so far have an extra 32-bit integer field in their TL description, normally equal to zero. However, if the key obtained by hashing that description cannot be retrieved from or updated in the TON DHT, the value in this field is increased, and

a new attempt is made. In this way, one cannot "capture" and "censor" a key (i.e., perform a key retention attack) by creating a lot of abstract addresses lying near the key under attack and controlling the corresponding DHT nodes.

3.2.12. Locating services. Some services, located in the TON Network and available through the (higher-level protocols built upon the) TON ADNL described in **3.1**, may want to publish their abstract addresses somewhere, so that their clients would know where to find them.

However, publishing the service's abstract address in the TON Blockchain may not be the best approach, because the abstract address might need to be changed quite often, and because it could make sense to provide several addresses, for reliability or load balancing purposes.

An alternative is to publish a public key into the TON Blockchain, and use a special DHT key indicating that public key as its "owner" in the TL description string (cf. **2.2.5**) to publish an up-to-date list of the service's abstract addresses. This is one of the approaches exploited by TON Services.

3.2.13. Locating owners of TON blockchain accounts. In most cases, owners of TON blockchain accounts would not like to be associated with abstract network addresses, and especially IP addresses, because this can violate their privacy. In some cases, however, the owner of a TON blockchain account may want to publish one or several abstract addresses where she could be contacted.

A typical case is that of a node in the TON Payments "lightning network" (cf. **5.2**), the platform for instant cryptocurrency transfers. A public TON Payments node may want not only to establish payment channels with other peers, but also to publish an abstract network address that could be used to contact it at a later time for transferring payments along the already-established channels.

One option would be to include an abstract network address in the smart contract creating the payment channel. A more flexible option is to include a public key in the smart contract, and then use DHT as explained in **3.2.12**.

The most natural way would be to use the same private key that controls the account in the TON Blockchain to sign and publish updates in the TON DHT about the abstract addresses associated with that account. This is done almost in the same way as described in **3.2.12**; however, the DHT key employed would require a special key description, containing only the

account_id itself, equal to SHA256 of the "account description", which contains the public key of the account. The signature, included in the value of this DHT key, would contain the account description as well.

In this way, a mechanism for locating abstract network addresses of some owners of the TON Blockchain accounts becomes available.

3.2.14. Locating abstract addresses. Notice that the TON DHT, while being implemented over TON ADNL, is itself used by the TON ADNL for several purposes.

The most important of them is to locate a node or its contact data starting from its 256-bit abstract address. This is necessary because the TON ADNL should be able to send datagrams to arbitrary 256-bit abstract addresses, even if no additional information is provided.

To this end, the 256-bit abstract address is simply looked up as a key in the DHT. Either a node with this address (i.e., using this address as a public semi-persistent DHT address) is found, in which case its IP address and port can be learned; or, an input tunnel description may be retrieved as the value of the key in question, signed by the correct private key, in which case this tunnel description would be used to send ADNL datagrams to the intended recipient.

Notice that in order to make an abstract address "public" (reachable from any nodes in the network), its owner must either use it as a semi-permanent DHT address, or publish (in the DHT key equal to the abstract address under consideration) an input tunnel description with another of its public abstract addresses (e.g., the semi-permanent address) as the tunnel's entry point. Another option would be to simply publish its IP address and UDP port.

3.3 Overlay Networks and Multicasting Messages

In a multi-blockchain system like the TON Blockchain, even full nodes would normally be interested in obtaining updates (i.e., new blocks) only about some shardchains. To this end, a special overlay (sub)network must be built inside the TON Network, on top of the ADNL protocol discussed in **3.1**, one for each shardchain.

Therefore, the need to build arbitrary overlay subnetworks, open to any nodes willing to participate, arises. Special gossip protocols, built upon ADNL, will be run in these overlay networks. In particular, these gossip

protocols may be used to propagate (broadcast) arbitrary data inside such a subnetwork.

3.3.1. Overlay networks. An *overlay (sub)network* is simply a (virtual) network implemented inside some larger network. Usually only some nodes of the larger network participate in the overlay subnetwork, and only some "links" between these nodes, physical or virtual, are part of the overlay subnetwork.

In this way, if the encompassing network is represented as a graph (perhaps a full graph in the case of a datagram network such as ADNL, where any node can easily communicate to any other), the overlay subnetwork is a *subgraph* of this graph.

In most cases, the overlay network is implemented using some protocol built upon the network protocol of the larger network. It may use the same addresses as the larger network, or use custom addresses.

3.3.2. Overlay networks in TON. Overlay networks in TON are built upon the ADNL protocol discussed in **3.1**; they use 256-bit ADNL abstract addresses as addresses in the overlay networks as well. Each node usually selects one of its abstract addresses to double as its address in the overlay network.

In contrast to ADNL, the TON overlay networks usually do not support sending datagrams to arbitrary other nodes. Instead, some "semipermanent links" are established between some nodes (called "neighbors" with respect to the overlay network under consideration), and messages are usually forwarded along these links (i.e., from a node to one of its neighbors). In this way, a TON overlay network is a (usually not full) subgraph inside the (full) graph of the ADNL network.

Links to neighbors in TON overlay networks can be implemented using dedicated peer-to-peer ADNL channels (cf. **3.1.5**).

Each node of an overlay network maintains a list of neighbors (with respect to the overlay network), containing their abstract addresses (which they use to identify them in the overlay network) and some link data (e.g., the ADNL channel used to communicate with them).

3.3.3. Private and public overlay networks. Some overlay networks are *public*, meaning that any node can join them at will. Other are *private*, meaning that only certain nodes can be admitted (e.g., those that can prove

their identities as validators.) Some private overlay networks can even be unknown to the "general public". The information about such overlay networks is made available only to certain trusted nodes; for example, it can be encrypted with a public key, and only nodes having a copy of the corresponding private key will be able to decrypt this information.

3.3.4. Centrally controlled overlay networks. Some overlay networks are *centrally controlled*, by one or several nodes, or by the owner of some widely-known public key. Others are *decentralized*, meaning that there are no specific nodes responsible for them.

3.3.5. Joining an overlay network. When a node wants to join an overlay network, it first must learn its 256-bit *network identifier*, usually equal to SHA256 of the *description* of the overlay network—a TL-serialized object (cf. **2.2.5**) which may contain, for instance, the central authority of the overlay network (i.e., its public key and perhaps its abstract address,[33]) a string with the name of the overlay network, a TON Blockchain shard identifier if this is an overlay network related to that shard, and so on.

Sometimes it is possible to recover the overlay network description starting from the network identifier, simply by looking it up in the TON DHT. In other cases (e.g., for private overlay networks), one must obtain the network description along with the network identifier.

3.3.6. Locating one member of the overlay network. After a node learns the network identifier and the network description of the overlay network it wants to join, it must locate at least one node belonging to that network.

This is also needed for nodes that do not want to join the overlay network, but want just to communicate with it; for example, there might be an overlay network dedicated to collecting and propagating transaction candidates for a specific shardchain, and a client might want to connect to any node of this network to suggest a transaction.

The method used for locating members of an overlay network is defined in the description of that network. Sometimes (especially for private networks) one must already know a member node to be able to join. In other cases, the abstract addresses of some nodes are contained in the network description. A more flexible approach is to indicate in the network description only the

[33] Alternatively, the abstract address might be stored in the DHT as explained in **3.2.12**.

central authority responsible for the network, and then the abstract addresses will be available through values of certain DHT keys, signed by that central authority.

Finally, truly decentralized public overlay networks can use the "distributed torrent-tracker" mechanism described in **3.2.10**, also implemented with the aid of the TON DHT.

3.3.7. Locating more members of the overlay network. Creating links. Once one node of the overlay network is found, a special query may be sent to that node requesting a list of other members, for instance, neighbors of the node being queried, or a random selection thereof.

This enables the joining member to populate her "adjacency" or "neighbor list" with respect to the overlay network, by selecting some newly-learned network nodes and establishing links to them (i.e., dedicated ADNL point-to-point channels, as outlined in **3.3.2**). After that, special messages are sent to all neighbors indicating that the new member is ready to work in the overlay network. The neighbors include their links to the new member in their neighbor lists.

3.3.8. Maintaining the neighbor list. An overlay network node must update its neighbor list from time to time. Some neighbors, or at least links (channels) to them, may stop responding; in this case, these links must be marked as "suspended", some attempts to reconnect to such neighbors must be made, and, if these attempts fail, the links must be destroyed.

On the other hand, every node sometimes requests from a randomly chosen neighbor its list of neighbors (or some random selection thereof), and uses it to partially update its own neighbor list, by adding some newly-discovered nodes to it, and removing some of the old ones, either randomly or depending on their response times and datagram loss statistics.

3.3.9. The overlay network is a random subgraph. In this way, the overlay network becomes a random subgraph inside the ADNL network. If the degree of each vertex is at least three (i.e., if each node is connected to at least three neighbors), this random graph is known to be *connected* with a probability almost equal to one. More precisely, the probability of a random graph with n vertices being *dis*connected is exponentially small, and this probability can be completely neglected if, say, $n \geq 20$. (Of course, this does not apply in the case of a global network partition, when nodes on different sides of the partition have no chance to learn about each other.) On the

other hand, if n is smaller than 20, it would suffice to require each vertex to have, say, at least ten neighbors.

3.3.10. TON overlay networks are optimized for lower latency. TON overlay networks optimize the "random" network graph generated by the previous method as follows. Every node tries to retain at least three neighbors with the minimal round-trip time, changing this list of "fast neighbors" very rarely. At the same time, it also has at least three other "slow neighbors" that are chosen completely randomly, so that the overlay network graph would always contain a random subgraph. This is required to maintain connectivity and prevent splitting of the overlay network into several unconnected regional subnetworks. At least three "intermediate neighbors", which have intermediate round-trip times, bounded by a certain constant (actually, a function of the round-trip times of the fast and the slow neighbors), are also chosen and retained.

In this way, the graph of an overlay network still maintains enough randomness to be connected, but is optimized for lower latency and higher throughput.

3.3.11. Gossip protocols in an overlay network. An overlay network is often used to run one of the so-called *gossip protocols*, which achieve some global goal while letting every node interact only with its neighbors. For example, there are gossip protocols to construct an approximate list of all members of a (not too large) overlay network, or to compute an estimate of the number of members of an (arbitrarily large) overlay network, using only a bounded amount of memory at each node (cf. [15, 4.4.3] or [1] for details).

3.3.12. Overlay network as a broadcast domain. The most important gossip protocol running in an overlay network is the *broadcast protocol*, intended to propagate broadcast messages generated by any node of the network, or perhaps by one of the designated sender nodes, to all other nodes.

There are in fact several broadcast protocols, optimized for different use cases. The simplest of them receives new broadcast messages and relays them to all neighbors that have not yet sent a copy of that message themselves.

3.3.13. More sophisticated broadcast protocols. Some applications may warrant more sophisticated broadcast protocols. For instance, for broadcasting messages of substantial size, it makes sense to send to the neighbors not the newly-received message itself, but its hash (or a collection of hashes

of new messages). The neighbor may request the message itself after learning a previously unseen message hash, to be transferred, say, using the reliable large datagram protocol (RLDP) discussed in **3.1.9**. In this way, the new message will be downloaded from one neighbor only.

3.3.14. Checking the connectivity of an overlay network. The connectivity of an overlay network can be checked if there is a known node (e.g., the "owner" or the "creator" of the overlay network) that must be in this overlay network. Then the node in question simply broadcasts from time to time short messages containing the current time, a sequence number and its signature. Any other node can be sure that it is still connected to the overlay network if it has received such a broadcast not too long ago. This protocol can be extended to the case of several well-known nodes; for example, they all will send such broadcasts, and all other nodes will expect to receive broadcasts from more than half of the well-known nodes.

In the case of an overlay network used for propagating new blocks (or just new block headers) of a specific shardchain, a good way for a node to check connectivity is to keep track of the most recent block received so far. Because a block is normally generated every five seconds, if no new block is received for more than, say, thirty seconds, the node probably has been disconnected from the overlay network.

3.3.15. Streaming broadcast protocol. Finally, there is a *streaming broadcast protocol* for TON overlay networks, used, for example, to propagate block candidates among validators of some shardchain ("shardchain task group"), who, of course, create a private overlay network for that purpose. The same protocol can be used to propagate new shardchain blocks to all full nodes for that shardchain.

This protocol has already been outlined in **2.6.10**: the new (large) broadcast message is split into, say, N one-kilobyte chunks; the sequence of these chunks is augmented to $M \geq N$ chunks by means of an erasure code such as the Reed–Solomon or a fountain code (e.g., the RaptorQ code [9] [14]), and these M chunks are streamed to all neighbors in ascending chunk number order. The participating nodes collect these chunks until they can recover the original large message (one would have to successfully receive at least N of the chunks for this), and then instruct their neighbors to stop sending new chunks of the stream, because now these nodes can generate the subsequent chunks on their own, having a copy of the original message. Such nodes continue to generate the subsequent chunks of the stream and send them to

their neighbors, unless the neighbors in turn indicate that this is no longer necessary.

In this way, a node does not need to download a large message in its entirety before propagating it further. This minimizes broadcast latency, especially when combined with the optimizations described in **3.3.10**.

3.3.16. Constructing new overlay networks based on existing ones. Sometimes one does not want to construct an overlay network from scratch. Instead, one or several previously existing overlay networks are known, and the membership of the new overlay network is expected to overlap significantly with the combined membership of these overlay networks.

An important example arises when a TON shardchain is split in two, or two sibling shardchains are merged into one (cf. **2.7**). In the first case, the overlay networks used for propagating new blocks to full nodes must be constructed for each of the new shardchains; however, each of these new overlay networks can be expected to be contained in the block propagation network of the original shardchain (and comprise approximately half its members). In the second case, the overlay network for propagating new blocks of the merged shardchain will consist approximately of the union of members of the two overlay networks related to the two sibling shardchains being merged.

In such cases, the description of the new overlay network may contain an explicit or implicit reference to a list of related existing overlay networks. A node wishing to join the new overlay network may check whether it is already a member of one of these existing networks, and query its neighbors in these networks whether they are interested in the new network as well. In case of a positive answer, new point-to-point channels can be established to such neighbors, and they can be included in the neighbor list for the new overlay network.

This mechanism does not totally supplant the general mechanism described in **3.3.6** and **3.3.7**; rather, both are run in parallel and are used to populate the neighbor list. This is needed to prevent inadvertent splitting of the new overlay network into several unconnected subnetworks.

3.3.17. Overlay networks within overlay networks. Another interesting case arises in the implementation of *TON Payments* (a "lightning network" for instant off-chain value transfers; cf. **5.2**). In this case, first an overlay network containing all transit nodes of the "lightning network" is constructed. However, some of these nodes have established payment channels in the blockchain; they must always be neighbors in this overlay network, in

addition to any "random" neighbors selected by the general overlay network algorithms described in **3.3.6**, **3.3.7** and **3.3.8**. These "permanent links" to the neighbors with established payment channels are used to run specific lightning network protocols, thus effectively creating an overlay subnetwork (not necessarily connected, if things go awry) inside the encompassing (almost always connected) overlay network.

4 TON Services and Applications

We have discussed the TON Blockchain and TON Networking technologies at some length. Now we explain some ways in which they can be combined to create a wide range of services and applications, and discuss some of the services that will be provided by the TON Project itself, either from the very beginning or at a later time.

4.1 TON Service Implementation Strategies

We start with a discussion of how different blockchain and network-related applications and services may be implemented inside the TON ecosystem. First of all, a simple classification is in order:

4.1.1. Applications and services. We will use the words "application" and "service" interchangeably. However, there is a subtle and somewhat vague distinction: an *application* usually provides some services directly to human users, while a *service* is usually exploited by other applications and services. For example, TON Storage is a service, because it is designed to keep files on behalf of other applications and services, even though a human user might use it directly as well. A hypothetical "Facebook in a blockchain" (cf. **2.9.13**) or Telegram messenger, if made available through the TON Network (i.e., implemented as a "ton-service"; cf. **4.1.6**), would rather be an *application*, even though some "bots" might access it automatically without human intervention.

4.1.2. Location of the application: on-chain, off-chain or mixed. A service or an application designed for the TON ecosystem needs to keep its data and process that data somewhere. This leads to the following classification of applications (and services):

- *On-chain* applications (cf. **4.1.4**): All data and processing are in the TON Blockchain.
- *Off-chain* applications (cf. **4.1.5**): All data and processing are outside the TON Blockchain, on servers available through the TON Network.
- *Mixed* applications (cf. **4.1.7**): Some, but not all, data and processing are in the TON Blockchain; the rest are on off-chain servers available through the TON Network.

4.1.3. Centralization: centralized and decentralized, or distributed, applications. Another classification criterion is whether the application (or service) relies on a centralized server cluster, or is really "distributed" (cf. **4.1.9**). All on-chain applications are automatically decentralized and distributed. Off-chain and mixed applications may exhibit different degrees of centralization.

Now let us consider the above possibilities in more detail.

4.1.4. Pure "on-chain" applications: distributed applications, or "dapps", residing in the blockchain. One of the possible approaches, mentioned in **4.1.2**, is to deploy a "distributed application" (commonly abbreviated as "dapp") completely in the TON Blockchain, as one smart contract or a collection of smart contracts. All data will be kept as part of the permanent state of these smart contracts, and all interaction with the project will be done by means of (TON Blockchain) messages sent to or received from these smart contracts.

We have already discussed in **2.9.13** that this approach has its drawbacks and limitations. It has its advantages, too: such a distributed application needs no servers to run on or to store its data (it runs "in the blockchain"—i.e., on the validators' hardware), and enjoys the blockchain's extremely high (Byzantine) reliability and accessibility. The developer of such a distributed application does not need to buy or rent any hardware; all she needs to do is develop some software (i.e., the code for the smart contracts). After that, she will effectively rent the computing power from the validators, and will pay for it in Grams, either herself or by putting this burden on the shoulders of her users.

4.1.5. Pure network services: "ton-sites" and "ton-services". Another extreme option is to deploy the service on some servers and make it available to the users through the ADNL protocol described in **3.1**, and maybe some higher level protocol such as the RLDP discussed in **3.1.9**, which can be used to send RPC queries to the service in any custom format and obtain answers to these queries. In this way, the service will be totally off-chain, and will reside in the TON Network, almost without using the TON Blockchain.

The TON Blockchain might be used only to locate the abstract address or addresses of the service, as outlined in **3.2.12**, perhaps with the aid of a service such as the TON DNS (cf. **4.3.1**) to facilitate translation of domain-

like human-readable strings into abstract addresses.

To the extent the ADNL network (i.e., the TON Network) is similar to the Invisible Internet Project (I^2P), such (almost) purely network services are analogous to the so-called "eep-services" (i.e., services that have an I^2P-address as their entry point, and are available to clients through the I^2P network). We will say that such purely network services residing in the TON Network are "ton-services".

An "eep-service" may implement HTTP as its client-server protocol; in the TON Network context, a "ton-service" might simply use RLDP (cf. **3.1.9**) datagrams to transfer HTTP queries and responses to them. If it uses the TON DNS to allow its abstract address to be looked up by a human-readable domain name, the analogy to a web site becomes almost perfect. One might even write a specialized browser, or a special proxy ("ton-proxy") that is run locally on a user's machine, accepts arbitrary HTTP queries from an ordinary web browser the user employs (once the local IP address and the TCP port of the proxy are entered into the browser's configuration), and forwards these queries through the TON Network to the abstract address of the service. Then the user would have a browsing experience similar to that of the World Wide Web (WWW).

In the I^2P ecosystem, such "eep-services" are called "eep-sites". One can easily create "ton-sites" in the TON ecosystem as well. This is facilitated somewhat by the existence of services such as the TON DNS, which exploit the TON Blockchain and the TON DHT to translate (TON) domain names into abstract addresses.

4.1.6. Telegram Messenger as a ton-service; MTProto over RLDP. We would like to mention in passing that the MTProto protocol,[34] used by Telegram Messenger[35] for client-server interaction, can be easily embedded into the RLDP protocol discussed in **3.1.9**, thus effectively transforming Telegram into a ton-service. Because the TON Proxy technology can be switched on transparently for the end user of a ton-site or a ton-service, being implemented on a lower level than the RLDP and ADNL protocols (cf. **3.1.6**), this would make Telegram effectively unblockable. Of course, other messaging and social networking services might benefit from this technology as well.

[34] `https://core.telegram.org/mtproto`

[35] `https://telegram.org/`

4.1.7. Mixed services: partly off-chain, partly on-chain. Some services might use a mixed approach: do most of the processing off-chain, but also have some on-chain part (for example, to register their obligations towards their users, and vice versa). In this way, part of the state would still be kept in the TON Blockchain (i.e., an immutable public ledger), and any misbehavior of the service or of its users could be punished by smart contracts.

4.1.8. Example: keeping files off-chain; TON Storage. An example of such a service is given by *TON Storage*. In its simplest form, it allows users to store files off-chain, by keeping on-chain only a hash of the file to be stored, and possibly a smart contract where some other parties agree to keep the file in question for a given period of time for a pre-negotiated fee. In fact, the file may be subdivided into chunks of some small size (e.g., 1 kilobyte), augmented by an erasure code such as a Reed–Solomon or a fountain code, a Merkle tree hash may be constructed for the augmented sequence of chunks, and this Merkle tree hash might be published in the smart contract instead of or along with the usual hash of the file. This is somewhat reminiscent of the way files are stored in a torrent.

An even simpler form of storing files is completely off-chain: one might essentially create a "torrent" for a new file, and use TON DHT as a "distributed torrent tracker" for this torrent (cf. **3.2.10**). This might actually work pretty well for popular files. However, one does not get any availability guarantees. For example, a hypothetical "blockchain Facebook" (cf. **2.9.13**), which would opt to keep the profile photographs of its users completely off-chain in such "torrents", might risk losing photographs of ordinary (not especially popular) users, or at least risk being unable to present these photographs for prolonged periods. The TON Storage technology, which is mostly off-chain, but uses an on-chain smart contract to enforce availability of the stored files, might be a better match for this task.

4.1.9. Decentralized mixed services, or "fog services". So far, we have discussed *centralized* mixed services and applications. While their on-chain component is processed in a decentralized and distributed fashion, being located in the blockchain, their off-chain component relies on some servers controlled by the service provider in the usual centralized fashion. Instead of using some dedicated servers, computing power might be rented from a cloud computing service offered by one of the large companies. However, this would not lead to decentralization of the off-chain component of the service.

A decentralized approach to implementing the off-chain component of a service consists in creating a *market*, where anybody possessing the required hardware and willing to rent their computing power or disk space would offer their services to those needing them.

For example, there might exist a registry (which might also be called a "market" or an "exchange") where all nodes interested in keeping files of other users publish their contact information, along with their available storage capacity, availability policy, and prices. Those needing these services might look them up there, and, if the other party agrees, create smart contracts in the blockchain and upload files for off-chain storage. In this way a service like *TON Storage* becomes truly decentralized, because it does not need to rely on any centralized cluster of servers for storing files.

4.1.10. Example: "fog computing" platforms as decentralized mixed services. Another example of such a decentralized mixed application arises when one wants to perform some specific computations (e.g., 3D rendering or training neural networks), often requiring specific and expensive hardware. Then those having such equipment might offer their services through a similar "exchange", and those needing such services would rent them, with the obligations of the sides registered by means of smart contracts. This is similar to what "fog computing" platforms, such as Golem (`https://golem.network/`) or SONM (`https://sonm.io/`), promise to deliver.

4.1.11. Example: TON Proxy is a fog service. *TON Proxy* provides yet another example of a fog service, where nodes wishing to offer their services (with or without compensation) as tunnels for ADNL network traffic might register, and those needing them might choose one of these nodes depending on the price, latency and bandwidth offered. Afterwards, one might use payment channels provided by *TON Payments* for processing micropayments for the services of those proxies, with payments collected, for instance, for every 128 KiB transferred.

4.1.12. Example: TON Payments is a fog service. The TON Payments platform (cf. **5**) is also an example of such a decentralized mixed application.

4.2 Connecting Users and Service Providers

We have seen in **4.1.9** that "fog services" (i.e., mixed decentralized services) will usually need some *markets*, *exchanges* or *registries*, where those needing

specific services might meet those providing them.

Such markets are likely to be implemented as on-chain, off-chain or mixed services themselves, centralized or distributed.

4.2.1. Example: connecting to TON Payments. For example, if one wants to use TON Payments (cf. **5**), the first step would be to find at least some existing transit nodes of the "lightning network" (cf. **5.2**), and establish payment channels with them, if they are willing. Some nodes can be found with the aid of the "encompassing" overlay network, which is supposed to contain all transit lightning network nodes (cf. **3.3.17**). However, it is not clear whether these nodes will be willing to create new payment channels. Therefore, a registry is needed where nodes ready to create new links can publish their contact information (e.g., their abstract addresses).

4.2.2. Example: uploading a file into TON Storage. Similarly, if one wants to upload a file into the TON Storage, she must locate some nodes willing to sign a smart contract binding them to keep a copy of that file (or of any file below a certain size limit, for that matter). Therefore, a registry of nodes offering their services for storing files is needed.

4.2.3. On-chain, mixed and off-chain registries. Such a registry of service providers might be implemented completely on-chain, with the aid of a smart contract which would keep the registry in its permanent storage. However, this would be quite slow and expensive. A mixed approach is more efficient, where the relatively small and rarely changed on-chain registry is used only to point out some nodes (by their abstract addresses, or by their public keys, which can be used to locate actual abstract addresses as described in **3.2.12**), which provide off-chain (centralized) registry services.

Finally, a decentralized, purely off-chain approach might consist of a public overlay network (cf. **3.3**), where those willing to offer their services, or those looking to buy somebody's services, simply broadcast their offers, signed by their private keys. If the service to be provided is very simple, even broadcasting the offers might be not necessary: the approximate membership of the overlay network itself might be used as a "registry" of those willing to provide a particular service. Then a client requiring this service might locate (cf. **3.3.7**) and query some nodes of this overlay network, and then query their neighbors, if the nodes already known are not ready to satisfy its needs.

4.2.4. Registry or exchange in a side-chain. Another approach to implementing decentralized mixed registries consists in creating an independent specialized blockchain ("side-chain"), maintained by its own set of self-proclaimed validators, who publish their identities in an on-chain smart contract and provide network access to all interested parties to this specialized blockchain, collecting transaction candidates and broadcasting block updates through dedicated overlay networks (cf. **3.3**). Then any full node for this sidechain can maintain its own copy of the shared registry (essentially equal to the global state of this side-chain), and process arbitrary queries related to this registry.

4.2.5. Registry or exchange in a workchain. Another option is to create a dedicated workchain inside the TON Blockchain, specialized for creating registries, markets and exchanges. This might be more efficient and less expensive than using smart contracts residing in the basic workchain (cf. **2.1.11**). However, this would still be more expensive than maintaining registries in side-chains (cf. **4.2.4**).

4.3 Accessing TON Services

We have discussed in **4.1** the different approaches one might employ for creating new services and applications residing in the TON ecosystem. Now we discuss how these services might be accessed, and some of the "helper services" that will be provided by TON, including *TON DNS* and *TON Storage*.

4.3.1. TON DNS: a mostly on-chain hierarchical domain name service. The *TON DNS* is a predefined service, which uses a collection of smart contracts to keep a map from human-readable domain names to (256-bit) addresses of ADNL network nodes and TON Blockchain accounts and smart contracts.

While anybody might in principle implement such a service using the TON Blockchain, it is useful to have such a predefined service with a well-known interface, to be used by default whenever an application or a service wants to translate human-readable identifiers into addresses.

4.3.2. TON DNS use cases. For example, a user looking to transfer some cryptocurrency to another user or to a merchant may prefer to remember a TON DNS domain name of the account of that user or merchant, instead of

keeping their 256-bit account identifiers at hand and copy-pasting them into the recipient field in their light wallet client.

Similarly, TON DNS may be used to locate account identifiers of smart contracts or entry points of ton-services and ton-sites (cf. **4.1.5**), enabling a specialized client ("ton-browser"), or a usual internet browser combined with a specialized ton-proxy extension or stand-alone application, to deliver a WWW-like browsing experience to the user.

4.3.3. TON DNS smart contracts. The TON DNS is implemented by means of a tree of special (DNS) smart contracts. Each DNS smart contract is responsible for registering subdomains of some fixed domain. The "root" DNS smart contract, where level one domains of the TON DNS system will be kept, is located in the masterchain. Its account identifier must be hardcoded into all software that wishes to access the TON DNS database directly.

Any DNS smart contract contains a hashmap, mapping variable-length null-terminated UTF-8 strings into their "values". This hashmap is implemented as a binary Patricia tree, similar to that described in **2.3.7** but supporting variable-length bitstrings as keys.

4.3.4. Values of the DNS hashmap, or TON DNS records. As to the values, they are "TON DNS records" described by a TL-scheme (cf. **2.2.5**). They consist of a "magic number", selecting one of the options supported, and then either an account identifier, or a smart-contract identifier, or an abstract network address (cf. **3.1**), or a public key used to locate abstract addresses of a service (cf. **3.2.12**), or a description of an overlay network, and so on. An important case is that of another DNS smart contract: in such a case, that smart contract is used to resolve subdomains of its domain. In this way, one can create separate registries for different domains, controlled by the owners of those domains.

These records may also contain an expiration time, a caching time (usually very large, because updating values in a blockchain too often is expensive), and in most cases a reference to the owner of the subdomain in question. The owner has the right to change this record (in particular, the owner field, thus transferring the domain to somebody else's control), and to prolong it.

4.3.5. Registering new subdomains of existing domains. In order to register a new subdomain of an existing domain, one simply sends a message to the smart contract, which is the registrar of that domain, containing the subdomain (i.e., the key) to be registered, the value in one of several prede-

fined formats, an identity of the owner, an expiration date, and some amount of cryptocurrency as determined by the domain's owner.

Subdomains are registered on a "first-come, first-served" basis.

4.3.6. Retrieving data from a DNS smart contract. In principle, any full node for the masterchain or shardchain containing a DNS smart contract might be able to look up any subdomain in the database of that smart contract, if the structure and the location of the hashmap inside the persistent storage of the smart contract are known.

However, this approach would work only for certain DNS smart contracts. It would fail miserably if a non-standard DNS smart contract were used.

Instead, an approach based on *general smart contract interfaces* and *get methods* (cf. **4.3.11**) is used. Any DNS smart contract must define a "get method" with a "known signature", which is invoked to look up a key. Since this approach makes sense for other smart contracts as well, especially those providing on-chain and mixed services, we explain it in some detail in **4.3.11**.

4.3.7. Translating a TON DNS domain. Once any full node, acting by itself or on behalf of some light client, can look up entries in the database of any DNS smart contract, arbitrary TON DNS domain names can be recursively translated, starting from the well-known and fixed root DNS smart contract (account) identifier.

For example, if one wants to translate `A.B.C`, one looks up keys `.C`, `.B.C`, and `A.B.C` in the root domain database. If the first of them is not found, but the second is, and its value is a reference to another DNS smart contract, then `A` is looked up in the database of that smart contract and the final value is retrieved.

4.3.8. Translating TON DNS domains for light nodes. In this way, a full node for the masterchain—and also for all shardchains involved in the domain look-up process—might translate any domain name into its current value without external help. A light node might request a full node to do this on its behalf and return the value, along with a Merkle proof (cf. **2.5.11**). This Merkle proof would enable the light node to verify that the answer is correct, so such TON DNS responses cannot be "spoofed" by a malicious interceptor, in contrast to the usual DNS protocol.

Because no node can be expected to be a full node with respect to all shardchains, actual TON DNS domain translation would involve a combination of these two strategies.

4.3.9. Dedicated "TON DNS servers". One might provide a simple "TON DNS server", which would receive RPC "DNS" queries (e.g., via the ADNL or RLDP protocols described in **3.1**), requesting that the server translate a given domain, process these queries by forwarding some subqueries to other (full) nodes if necessary, and return answers to the original queries, augmented by Merkle proofs if required.

Such "DNS servers" might offer their services (for free or not) to any other nodes and especially light clients, using one of the methods described in **4.2**. Notice that these servers, if considered part of the TON DNS service, would effectively transform it from a distributed on-chain service into a distributed mixed service (i.e., a "fog service").

This concludes our brief overview of the TON DNS service, a scalable on-chain registry for human-readable domain names of TON Blockchain and TON Network entities.

4.3.10. Accessing data kept in smart contracts. We have already seen that it is sometimes necessary to access data stored in a smart contract without changing its state.

If one knows the details of the smart-contract implementation, one can extract all the needed information from the smart contract's persistent storage, available to all full nodes of the shardchain the smart contract resides in. However, this is quite an inelegant way of doing things, depending very much on the smart-contract implementation.

4.3.11. "Get methods" of smart contracts. A better way would be to define some *get methods* in the smart contract, that is, some types of inbound messages that do not affect the state of the smart contract when delivered, but generate one or more output messages containing the "result" of the get method. In this way, one can obtain data from a smart contract, knowing only that it implements a get method with a known signature (i.e., a known format of the inbound message to be sent and outbound messages to be received as a result).

This way is much more elegant and in line with object oriented programming (OOP). However, it has an obvious defect so far: one must actually commit a transaction into the blockchain (sending the get message to the smart contract), wait until it is committed and processed by the validators, extract the answer from a new block, and pay for gas (i.e., for executing the get method on the validators' hardware). This is a waste of resources: get

methods do not change the state of the smart contract anyways, so they need not be executed in the blockchain.

4.3.12. Tentative execution of get methods of smart contracts. We have already remarked (cf. **2.4.6**) that any full node can tentatively execute any method of any smart contract (i.e., deliver any message to a smart contract), starting from a given state of the smart contract, without actually committing the corresponding transaction. The full node can simply load the code of the smart contract under consideration into the TON VM, initialize its persistent storage from the global state of the shardchain (known to all full nodes of the shardchain), and execute the smart-contract code with the inbound message as its input parameter. The output messages created will yield the result of this computation.

In this way, any full node can evaluate arbitrary get methods of arbitrary smart contracts, provided their signature (i.e., the format of inbound and outbound messages) is known. The node may keep track of the cells of the shardchain state accessed during this evaluation, and create a Merkle proof of the validity of the computation performed, for the benefit of a light node that might have asked the full node to do so (cf. **2.5.11**).

4.3.13. Smart-contract interfaces in TL-schemes. Recall that the methods implemented by a smart contract (i.e., the input messages accepted by it) are essentially some TL-serialized objects, which can be described by a TL-scheme (cf. **2.2.5**). The resulting output messages can be described by the same TL-scheme as well. In this way, the interface provided by a smart contract to other accounts and smart contracts may be formalized by means of a TL-scheme.

In particular, (a subset of) get methods supported by a smart contract can be described by such a formalized smart-contract interface.

4.3.14. Public interfaces of a smart contract. Notice that a formalized smart-contract interface, either in form of a TL-scheme (represented as a TL source file; cf. **2.2.5**) or in serialized form,[36] can be published—for example, in a special field in the smart-contract account description, stored in the blockchain, or separately, if this interface will be referred to many times. In the latter case a hash of the supported public interface may be incorporated into the smart-contract description instead of the interface description itself.

[36]TL-schemes can be TL-serialized themselves; cf. `https://core.telegram.org/mtproto/TL-tl`.

An example of such a public interface is that of a DNS smart contract, which is supposed to implement at least one standard get method for looking up subdomains (cf. **4.3.6**). A standard method for registering new subdomains can be also included in the standard public interface of DNS smart contracts.

4.3.15. User interface of a smart contract. The existence of a public interface for a smart contract has other benefits, too. For example, a wallet client application may download such an interface while examining a smart contract on the request of a user, and display a list of public methods (i.e., of available actions) supported by the smart contract, perhaps with some human-readable comments if any are provided in the formal interface. After the user selects one of these methods, a form may be automatically generated according to the TL-scheme, where the user will be prompted for all fields required by the chosen method and for the desired amount of cryptocurrency (e.g., Grams) to be attached to this request. Submitting this form will create a new blockchain transaction containing the message just composed, sent from the user's blockchain account.

In this way, the user will be able to interact with arbitrary smart contracts from the wallet client application in a user-friendly way by filling and submitting certain forms, provided these smart contracts have published their interfaces.

4.3.16. User interface of a "ton-service". It turns out that "ton-services" (i.e., services residing in the TON Network and accepting queries through the ADNL and RLDP protocols of **3**; cf. **4.1.5**) may also profit from having public interfaces, described by TL-schemes (cf. **2.2.5**). A client application, such as a light wallet or a "ton-browser", might prompt the user to select one of the methods and to fill in a form with parameters defined by the interface, similarly to what has just been discussed in **4.3.15**. The only difference is that the resulting TL-serialized message is not submitted as a transaction in the blockchain; instead, it is sent as an RPC query to the abstract address of the "ton-service" in question, and the response to this query is parsed and displayed according to the formal interface (i.e., a TL-scheme).

4.3.17. Locating user interfaces via TON DNS. The TON DNS record containing an abstract address of a ton-service or a smart-contract account identifier might also contain an optional field describing the public (user) interface of that entity, or several supported interfaces. Then the client

application (be it a wallet, a ton-browser or a ton-proxy) will be able to download the interface and interact with the entity in question (be it a smart contract or a ton-service) in a uniform way.

4.3.18. Blurring the distinction between on-chain and off-chain services. In this way, the distinction between on-chain, off-chain and mixed services (cf. **4.1.2**) is blurred for the end user: she simply enters the domain name of the desired service into the address line of her ton-browser or wallet, and the rest is handled seamlessly by the client application.

4.3.19. A light wallet and TON entity explorer can be built into Telegram Messenger clients. An interesting opportunity arises at this point. A light wallet and TON entity explorer, implementing the above functionality, can be embedded into the Telegram Messenger smartphone client application, thus bringing the technology to more than 200 million people. Users would be able to send hyperlinks to TON entities and resources by including TON URIs (cf. **4.3.22**) in messages; such hyperlinks, if selected, will be opened internally by the Telegram client application of the receiving party, and interaction with the chosen entity will begin.

4.3.20. "ton-sites" as ton-services supporting an HTTP interface. A *ton-site* is simply a ton-service that supports an HTTP interface, perhaps along with some other interfaces. This support may be announced in the corresponding TON DNS record.

4.3.21. Hyperlinks. Notice that the HTML pages returned by ton-sites may contain *ton-hyperlinks*—that is, references to other ton-sites, smart contracts and accounts by means of specially crafted URI schemes (cf. **4.3.22**)—containing either abstract network addresses, account identifiers, or human-readable TON DNS domains. Then a "ton-browser" might follow such a hyperlink when the user selects it, detect the interface to be used, and display a user interface form as outlined in **4.3.15** and **4.3.16**.

4.3.22. Hyperlink URLs may specify some parameters. The hyperlink URLs may contain not only a (TON) DNS domain or an abstract address of the service in question, but also the name of the method to be invoked and some or all of its parameters. A possible URI scheme for this might look as follows:

`ton://`*<domain>*`/`*<method>*`?`*<field1>*`=`*<value1>*`&`*<field2>*`=...`

When the user selects such a link in a ton-browser, either the action is performed immediately (especially if it is a get method of a smart contract, invoked anonymously), or a partially filled form is displayed, to be explicitly confirmed and submitted by the user (this may be required for payment forms).

4.3.23. POST actions. A ton-site may embed into the HTML pages it returns some usual-looking POST forms, with POST actions referring either to ton-sites, ton-services or smart contracts by means of suitable (TON) URLs. In that case, once the user fills and submits that custom form, the corresponding action is taken, either immediately or after an explicit confirmation.

4.3.24. TON WWW. All of the above will lead to the creation of a whole web of cross-referencing entities, residing in the TON Network, which would be accessible to the end user through a ton-browser, providing the user with a WWW-like browsing experience. For end users, this will finally make blockchain applications fundamentally similar to the web sites they are already accustomed to.

4.3.25. Advantages of TON WWW. This "TON WWW" of on-chain and off-chain services has some advantages over its conventional counterpart. For example, payments are inherently integrated in the system. User identity can be always presented to the services (by means of automatically generated signatures on the transactions and RPC requests generated), or hidden at will. Services would not need to check and re-check user credentials; these credentials can be published in the blockchain once and for all. User network anonymity can be easily preserved by means of TON Proxy, and all services will be effectively unblockable. Micropayments are also possible and easy, because ton-browsers can be integrated with the TON Payments system.

5 TON Payments

The last component of the TON Project we will briefly discuss in this text is *TON Payments*, the platform for (micro)payment channels and "lightning network" value transfers. It would enable "instant" payments, without the need to commit all transactions into the blockchain, pay the associated transaction fees (e.g., for the gas consumed), and wait five seconds until the block containing the transactions in question is confirmed.

The overall overhead of such instant payments is so small that one can use them for micropayments. For example, a TON file-storing service might charge the user for every 128 KiB of downloaded data, or a paid TON Proxy might require some tiny micropayment for every 128 KiB of traffic relayed.

While *TON Payments* is likely to be released later than the core components of the TON Project, some considerations need to be made at the very beginning. For example, the TON Virtual Machine (TON VM; cf. **2.1.20**), used to execute the code of TON Blockchain smart contracts, must support some special operations with Merkle proofs. If such support is not present in the original design, adding it at a later stage might become problematic (cf. **2.8.16**). We will see, however, that the TON VM comes with natural support for "smart" payment channels (cf. **5.1.9**) out of the box.

5.1 Payment Channels

We start with a discussion of point-to-point payment channels, and how they can be implemented in the TON Blockchain.

5.1.1. The idea of a payment channel. Suppose two parties, A and B, know that they will need to make a lot of payments to each other in the future. Instead of committing each payment as a transaction in the blockchain, they create a shared "money pool" (or perhaps a small private bank with exactly two accounts), and contribute some funds to it: A contributes a coins, and B contributes b coins. This is achieved by creating a special smart contract in the blockchain, and sending the money to it.

Before creating the "money pool", the two sides agree to a certain protocol. They will keep track of the *state* of the pool—that is, of their balances in the shared pool. Originally, the state is (a, b), meaning that a coins actually belong to A, and b coins belong to B. Then, if A wants to pay d coins to B, they can simply agree that the new state is $(a', b') = (a - d, b + d)$.

Afterwards, if, say, B wants to pay d' coins to A, the state will become $(a'', b'') = (a' + d', b' - d')$, and so on.

All this updating of balances inside the pool is done completely off-chain. When the two parties decide to withdraw their due funds from the pool, they do so according to the final state of the pool. This is achieved by sending a special message to the smart contract, containing the agreed-upon final state (a^*, b^*) along with the signatures of both A and B. Then the smart contract sends a^* coins to A, b^* coins to B and self-destructs.

This smart contract, along with the network protocol used by A and B to update the state of the pool, is a simple *payment channel between A and B*. According to the classification described in **4.1.2**, it is a *mixed* service: part of its state resides in the blockchain (the smart contract), but most of its state updates are performed off-chain (by the network protocol). If everything goes well, the two parties will be able to perform as many payments to each other as they want (with the only restriction being that the "capacity" of the channel is not overrun—i.e., their balances in the payment channel both remain non-negative), committing only two transactions into the blockchain: one to open (create) the payment channel (smart contract), and another to close (destroy) it.

5.1.2. Trustless payment channels. The previous example was somewhat unrealistic, because it assumes that both parties are willing to cooperate and will never cheat to gain some advantage. Imagine, for example, that A will choose not to sign the final balance (a', b') with $a' < a$. This would put B in a difficult situation.

To protect against such scenarios, one usually tries to develop *trustless* payment channel protocols, which do not require the parties to trust each other, and make provisions for punishing any party who would attempt to cheat.

This is usually achieved with the aid of signatures. The payment channel smart contract knows the public keys of A and B, and it can check their signatures if needed. The payment channel protocol requires the parties to sign the intermediate states and send the signatures to each other. Then, if one of the parties cheats—for instance, pretends that some state of the payment channel never existed—its misbehavior can be proved by showing its signature on that state. The payment channel smart contract acts as an "on-chain arbiter", able to process complaints of the two parties about each other, and punish the guilty party by confiscating all of its money and

awarding it to the other party.

5.1.3. Simple bidirectional synchronous trustless payment channel. Consider the following, more realistic example: Let the state of the payment channel be described by triple (δ_i, i, o_i), where i is the sequence number of the state (it is originally zero, and then it is increased by one when a subsequent state appears), δ_i is the *channel imbalance* (meaning that A and B own $a+\delta_i$ and $b-\delta_i$ coins, respectively), and o_i is the party allowed to generate the next state (either A or B). Each state must be signed both by A and B before any further progress can be made.

Now, if A wants to transfer d coins to B inside the payment channel, and the current state is $S_i = (\delta_i, i, o_i)$ with $o_i = A$, then it simply creates a new state $S_{i+1} = (\delta_i - d, i+1, o_{i+1})$, signs it, and sends it to B along with its signature. Then B confirms it by signing and sending a copy of its signature to A. After that, both parties have a copy of the new state with both of their signatures, and a new transfer may occur.

If A wants to transfer coins to B in a state S_i with $o_i = B$, then it first asks B to commit a subsequent state S_{i+1} with the same imbalance $\delta_{i+1} = \delta_i$, but with $o_{i+1} = A$. After that, A will be able to make its transfer.

When the two parties agree to close the payment channel, they both put their special *final* signatures on the state S_k they believe to be final, and invoke the *clean* or *two-sided finalization method* of the payment channel smart contract by sending it the final state along with both final signatures.

If the other party does not agree to provide its final signature, or simply if it stops responding, it is possible to close the channel unilaterally. For this, the party wishing to do so will invoke the *unilateral finalization* method, sending to the smart contract its version of the final state, its final signature, and the most recent state having a signature of the other party. After that, the smart contract does not immediately act on the final state received. Instead, it waits for a certain period of time (e.g., one day) for the other party to present its version of the final state. When the other party submits its version and it turns out to be compatible with the already submitted version, the "true" final state is computed by the smart contract and used to distribute the money accordingly. If the other party fails to present its version of the final state to the smart contract, then the money is redistributed according to the only copy of the final state presented.

If one of the two parties cheats—for example, by signing two different states as final, or by signing two different next states S_{i+1} and S'_{i+1}, or by

signing an invalid new state S_{i+1} (e.g., with imbalance $\delta_{i+1} < -a$ or $> b$)—then the other party may submit proof of this misbehavior to a third method of the smart contract. The guilty party is punished immediately by losing its share in the payment channel completely.

This simple payment channel protocol is *fair* in the sense that any party can always get its due, with or without the cooperation of the other party, and is likely to lose all of its funds committed to the payment channel if it tries to cheat.

5.1.4. Synchronous payment channel as a simple virtual blockchain with two validators. The above example of a simple synchronous payment channel can be recast as follows. Imagine that the sequence of states S_0, S_1, ..., S_n is actually the sequence of blocks of a very simple blockchain. Each block of this blockchain contains essentially only the current state of the blockchain, and maybe a reference to the previous block (i.e., its hash). Both parties A and B act as validators for this blockchain, so every block must collect both of their signatures. The state S_i of the blockchain defines the designated producer o_i for the next block, so there is no race between A and B for producing the next block. Producer A is allowed to create blocks that transfer funds from A to B (i.e., decrease the imbalance: $\delta_{i+1} \leq \delta_i$), and B can only transfer funds from B to A (i.e., increase δ).

If the two validators agree on the final block (and the final state) of the blockchain, it is finalized by collecting special "final" signatures of the two parties, and submitting them along with the final block to the channel smart contract for processing and re-distributing the money accordingly.

If a validator signs an invalid block, or creates a fork, or signs two different final blocks, it can be punished by presenting a proof of its misbehavior to the smart contract, which acts as an "on-chain arbiter" for the two validators; then the offending party will lose all its money kept in the payment channel, which is analogous to a validator losing its stake.

5.1.5. Asynchronous payment channel as a virtual blockchain with two workchains. The synchronous payment channel discussed in **5.1.3** has a certain disadvantage: one cannot begin the next transaction (money transfer inside the payment channel) before the previous one is confirmed by the other party. This can be fixed by replacing the single virtual blockchain discussed in **5.1.4** by a system of two interacting virtual workchains (or rather shardchains).

The first of these workchains contains only transactions by A, and its blocks can be generated only by A; its states are $S_i = (i, \phi_i, j, \psi_j)$, where i is the block sequence number (i.e., the count of transactions, or money transfers, performed by A so far), ϕ_i is the total amount transferred from A to B so far, j is the sequence number of the most recent valid block in B's blockchain that A is aware of, and ψ_j is the amount of money transferred from B to A in its j transactions. A signature of B put onto its j-th block should also be a part of this state. Hashes of the previous block of this workchain and of the j-th block of the other workchain may be also included. Validity conditions for S_i include $\phi_i \geq 0$, $\phi_i \geq \phi_{i-1}$ if $i > 0$, $\psi_j \geq 0$, and $-a \leq \psi_j - \phi_i \leq b$.

Similarly, the second workchain contains only transactions by B, and its blocks are generated only by B; its states are $T_j = (j, \psi_j, i, \phi_i)$, with similar validity conditions.

Now, if A wants to transfer some money to B, it simply creates a new block in its workchain, signs it, and sends to B, without waiting for confirmation.

The payment channel is finalized by A signing (its version of) the final state of its blockchain (with its special "final signature"), B signing the final state of its blockchain, and presenting these two final states to the clean finalization method of the payment channel smart contract. Unilateral finalization is also possible, but in that case the smart contract will have to wait for the other party to present its version of the final state, at least for some grace period.

5.1.6. Unidirectional payment channels. If only A needs to make payments to B (e.g., B is a service provider, and A its client), then a unilateral payment channel can be created. Essentially, it is just the first workchain described in **5.1.5** without the second one. Conversely, one can say that the asynchronous payment channel described in **5.1.5** consists of two unidirectional payment channels, or "half-channels", managed by the same smart contract.

5.1.7. More sophisticated payment channels. Promises. We will see later in **5.2.4** that the "lightning network" (cf. **5.2**), which enables instant money transfers through chains of several payment channels, requires higher degrees of sophistication from the payment channels involved.

In particular, we want to be able to commit "promises", or "conditional money transfers": A agrees to send c coins to B, but B will get the money

only if a certain condition is fulfilled, for instance, if B can present some string u with $\textsc{Hash}(u) = v$ for a known value of v. Otherwise, A can get the money back after a certain period of time.

Such a promise could easily be implemented on-chain by a simple smart contract. However, we want promises and other kinds of conditional money transfers to be possible off-chain, in the payment channel, because they considerably simplify money transfers along a chain of payment channels existing in the "lightning network" (cf. **5.2.4**).

The "payment channel as a simple blockchain" picture outlined in **5.1.4** and **5.1.5** becomes convenient here. Now we consider a more complicated virtual blockchain, the state of which contains a set of such unfulfilled "promises", and the amount of funds locked in such promises. This blockchain—or the two workchains in the asynchronous case—will have to refer explicitly to the previous blocks by their hashes. Nevertheless, the general mechanism remains the same.

5.1.8. Challenges for the sophisticated payment channel smart contracts. Notice that, while the final state of a sophisticated payment channel is still small, and the "clean" finalization is simple (if the two sides have agreed on their amounts due, and both have signed their agreement, nothing else remains to be done), the unilateral finalization method and the method for punishing fraudulent behavior need to be more complex. Indeed, they must be able to accept Merkle proofs of misbehavior, and to check whether the more sophisticated transactions of the payment channel blockchain have been processed correctly.

In other words, the payment channel smart contract must be able to work with Merkle proofs, to check their "hash validity", and must contain an implementation of *ev_trans* and *ev_block* functions (cf. **2.2.6**) for the payment channel (virtual) blockchain.

5.1.9. TON VM support for "smart" payment channels. The TON VM, used to run the code of TON Blockchain smart contracts, is up to the challenge of executing the smart contracts required for "smart", or sophisticated, payment channels (cf. **5.1.8**).

At this point the "everything is a bag of cells" paradigm (cf. **2.5.14**) becomes extremely convenient. Since all blocks (including the blocks of the ephemeral payment channel blockchain) are represented as bags of cells (and described by some algebraic data types), and the same holds for messages and Merkle proofs as well, a Merkle proof can easily be embedded into an

inbound message sent to the payment channel smart contract. The "hash condition" of the Merkle proof will be checked automatically, and when the smart contract accesses the "Merkle proof" presented, it will work with it as if it were a value of the corresponding algebraic data type—albeit incomplete, with some subtrees of the tree replaced by special nodes containing the Merkle hash of the omitted subtree. Then the smart contract will work with that value, which might represent, for instance, a block of the payment channel (virtual) blockchain along with its state, and will evaluate the *ev_block* function (cf. **2.2.6**) of that blockchain on this block and the previous state. Then either the computation finishes, and the final state can be compared with that asserted in the block, or an "absent node" exception is thrown while attempting to access an absent subtree, indicating that the Merkle proof is invalid.

In this way, the implementation of the verification code for smart payment channel blockchains turns out to be quite straightforward using TON Blockchain smart contracts. One might say that *the TON Virtual Machine comes with built-in support for checking the validity of other simple blockchains.* The only limiting factor is the size of the Merkle proof to be incorporated into the inbound message to the smart contract (i.e., into the transaction).

5.1.10. Simple payment channel within a smart payment channel. We would like to discuss the possibility of creating a simple (synchronous or asynchronous) payment channel inside an existing payment channel.

While this may seem somewhat convoluted, it is not much harder to understand and implement than the "promises" discussed in **5.1.7**. Essentially, instead of promising to pay c coins to the other party if a solution to some hash problem is presented, A promises to pay up to c coins to B according to the final settlement of some other (virtual) payment channel blockchain. Generally speaking, this other payment channel blockchain need not even be between A and B; it might involve some other parties, say, C and D, willing to commit c and d coins into their simple payment channel, respectively. (This possibility is exploited later in **5.2.5**.)

If the encompassing payment channel is asymmetric, two promises need to be committed into the two workchains: A will promise to pay $-\delta$ coins to B if the final settlement of the "internal" simple payment channel yields a negative final imbalance δ with $0 \leq -\delta \leq c$; and B will have to promise to pay δ to A if δ is positive. On the other hand, if the encompassing

payment channel is symmetric, this can be done by committing a single "simple payment channel creation" transaction with parameters (c, d) into the single payment channel blockchain by A (which would freeze c coins belonging to A), and then committing a special "confirmation transaction" by B (which would freeze d coins of B).

We expect the internal payment channel to be extremely simple (e.g., the simple synchronous payment channel discussed in **5.1.3**), to minimize the size of Merkle proofs to be submitted. The external payment channel will have to be "smart" in the sense described in **5.1.7**.

5.2 Payment Channel Network, or "Lightning Network"

Now we are ready to discuss the "lightning network" of TON Payments that enables instant money transfers between any two participating nodes.

5.2.1. Limitations of payment channels. A payment channel is useful for parties who expect a lot of money transfers between them. However, if one needs to transfer money only once or twice to a particular recipient, creating a payment channel with her would be impractical. Among other things, this would imply freezing a significant amount of money in the payment channel, and would require at least two blockchain transactions anyway.

5.2.2. Payment channel networks, or "lightning networks". Payment channel networks overcome the limitations of payment channels by enabling money transfers along *chains* of payment channels. If A wants to transfer money to E, she does not need to establish a payment channel with E. It would be sufficient to have a chain of payment channels linking A with E through several intermediate nodes—say, four payment channels: from A to B, from B to C, from C to D and from D to E.

5.2.3. Overview of payment channel networks. Recall that a *payment channel network*, known also as a "lightning network", consists of a collection of participating nodes, some of which have established long-lived payment channels between them. We will see in a moment that these payment channels will have to be "smart" in the sense of **5.1.7**. When a participating node A wants to transfer money to any other participating node E, she tries to find a path linking A to E inside the payment channel network. When such a path is found, she performs a "chain money transfer" along this path.

5.2.4. Chain money transfers. Suppose that there is a chain of payment channels from A to B, from B to C, from C to D, and from D to E. Suppose, further, that A wants to transfer x coins to E.

A simplistic approach would be to transfer x coins to B along the existing payment channel, and ask him to forward the money further to C. However, it is not evident why B would not simply take the money for himself. Therefore, one must employ a more sophisticated approach, not requiring all parties involved to trust each other.

This can be achieved as follows. A generates a large random number u and computes its hash $v = \textsc{Hash}(u)$. Then she creates a promise to pay x coins to B if a number u with hash v is presented (cf. **5.1.7**), inside her payment channel with B. This promise contains v, but not u, which is still kept secret.

After that, B creates a similar promise to C in their payment channel. He is not afraid to give such a promise, because he is aware of the existence of a similar promise given to him by A. If C ever presents a solution of the hash problem to collect x coins promised by B, then B will immediately submit this solution to A to collect x coins from A.

Then similar promises of C to D and of D to E are created. When the promises are all in place, A triggers the transfer by communicating the solution u to all parties involved—or just to E.

Some minor details are omitted in this description. For example, these promises must have different expiration times, and the amount promised might slightly differ along the chain (B might promise only $x - \epsilon$ coins to C, where ϵ is a small pre-agreed transit fee). We ignore such details for the time being, because they are not too relevant for understanding how payment channels work and how they can be implemented in TON.

5.2.5. Virtual payment channels inside a chain of payment channels. Now suppose that A and E expect to make a lot of payments to each other. They might create a new payment channel between them in the blockchain, but this would still be quite expensive, because some funds would be locked in this payment channel. Another option would be to use chain money transfers described in **5.2.4** for each payment. However, this would involve a lot of network activity and a lot of transactions in the virtual blockchains of all payment channels involved.

An alternative is to create a virtual payment channel inside the chain linking A to E in the payment channel network. For this, A and E create

a (virtual) blockchain for their payments, as if they were going to create a payment channel in the blockchain. However, instead of creating a payment channel smart contract in the blockchain, they ask all intermediate payment channels—those linking A to B, B to C, etc.—to create simple payment channels inside them, bound to the virtual blockchain created by A and E (cf. **5.1.10**). In other words, now a promise to transfer money according to the final settlement between A and E exists inside every intermediate payment channel.

If the virtual payment channel is unidirectional, such promises can be implemented quite easily, because the final imbalance δ is going to be non-positive, so simple payment channels can be created inside intermediate payment channels in the same order as described in **5.2.4**. Their expiration times can also be set in the same way.

If the virtual payment channel is bidirectional, the situation is slightly more complicated. In that case, one should split the promise to transfer δ coins according to the final settlement into two half-promises, as explained in **5.1.10**: to transfer $\delta^- = \max(0, -\delta)$ coins in the forward direction, and to transfer $\delta^+ = \max(0, \delta)$ in the backward direction. These half-promises can be created in the intermediate payment channels independently, one chain of half-promises in the direction from A to E, and the other chain in the opposite direction.

5.2.6. Finding paths in the lightning network. One point remains undiscussed so far: how will A and E find a path connecting them in the payment network? If the payment network is not too large, an OSPF-like protocol can be used: all nodes of the payment network create an overlay network (cf. **3.3.17**), and then every node propagates all available link (i.e., participating payment channel) information to its neighbors by a gossip protocol. Ultimately, all nodes will have a complete list of all payment channels participating in the payment network, and will be able to find the shortest paths by themselves—for example, by applying a version of Dijkstra's algorithm modified to take into account the "capacities" of the payment channels involved (i.e., the maximal amounts that can be transferred along them). Once a candidate path is found, it can be probed by a special ADNL datagram containing the full path, and asking each intermediate node to confirm the existence of the payment channel in question, and to forward this datagram further according to the path. After that, a chain can be constructed, and a protocol for chain transfers (cf. **5.2.4**), or for creating a

virtual payment channel inside a chain of payment channels (cf. **5.2.5**), can be run.

5.2.7. Optimizations. Some optimizations might be done here. For example, only transit nodes of the lightning network need to participate in the OSPF-like protocol discussed in **5.2.6**. Two "leaf" nodes wishing to connect through the lightning network would communicate to each other the lists of transit nodes they are connected to (i.e., with which they have established payment channels participating in the payment network). Then paths connecting transit nodes from one list to transit nodes from the other list can be inspected as outlined above in **5.2.6**.

5.2.8. Conclusion. We have outlined how the blockchain and network technologies of the TON project are adequate to the task of creating *TON Payments*, a platform for off-chain instant money transfers and micropayments. This platform can be extremely useful for services residing in the TON ecosystem, allowing them to easily collect micropayments when and where required.

Conclusion

We have proposed a scalable multi-blockchain architecture capable of supporting a massively popular cryptocurrency and decentralized applications with user-friendly interfaces.

To achieve the necessary scalability, we proposed the *TON Blockchain*, a "tightly-coupled" multi-blockchain system (cf. **2.8.14**) with bottom-up approach to sharding (cf. **2.8.12** and **2.1.2**). To further increase potential performance, we introduced the 2-blockchain mechanism for replacing invalid blocks (cf. **2.1.17**) and Instant Hypercube Routing for faster communication between shards (cf. **2.4.20**). A brief comparison of the TON Blockchain to existing and proposed blockchain projects (cf. **2.8** and **2.9**) highlights the benefits of this approach for systems that seek to handle millions of transactions per second.

The *TON Network*, described in Chapter **3**, covers the networking demands of the proposed multi-blockchain infrastructure. This network component may also be used in combination with the blockchain to create a wide spectrum of applications and services, impossible using the blockchain alone (cf. **2.9.13**). These services, discussed in Chapter **4**, include *TON DNS*, a service for translating human-readable object identifiers into their addresses; *TON Storage*, a distributed platform for storing arbitrary files; *TON Proxy*, a service for anonymizing network access and accessing TON-powered services; and *TON Payments* (cf. Chapter **5**), a platform for instant off-chain money transfers across the TON ecosystem that applications may use for micropayments.

The TON infrastructure allows for specialized light client wallet and "ton-browser" desktop and smartphone applications that enable a browser-like experience for the end user (cf. **4.3.24**), making cryptocurrency payments and interaction with smart contracts and other services on the TON Platform accessible to the mass user. Such a light client can be integrated into the Telegram Messenger client (cf. **4.3.19**), thus eventually bringing a wealth of blockchain-based applications to hundreds of millions of users.

References

[1] K. Birman, *Reliable Distributed Systems: Technologies, Web Services and Applications*, Springer, 2005.

[2] V. Buterin, *Ethereum: A next-generation smart contract and decentralized application platform*, `https://github.com/ethereum/wiki/wiki/White-Paper`, 2013.

[3] M. Ben-Or, B. Kelmer, T. Rabin, *Asynchronous secure computations with optimal resilience*, in *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, p. 183–192. ACM, 1994.

[4] M. Castro, B. Liskov, et al., *Practical byzantine fault tolerance*, *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (1999), p. 173–186, available at `http://pmg.csail.mit.edu/papers/osdi99.pdf`.

[5] EOS.IO, *EOS.IO technical white paper*, `https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md`, 2017.

[6] D. Goldschlag, M. Reed, P. Syverson, *Onion Routing for Anonymous and Private Internet Connections*, *Communications of the ACM*, **42**, num. 2 (1999), `http://www.onion-router.net/Publications/CACM-1999.pdf`.

[7] L. Lamport, R. Shostak, M. Pease, *The byzantine generals problem*, *ACM Transactions on Programming Languages and Systems*, **4/3** (1982), p. 382–401.

[8] S. Larimer, *The history of BitShares*, `https://docs.bitshares.org/bitshares/history.html`, 2013.

[9] M. Luby, A. Shokrollahi, et al., *RaptorQ forward error correction scheme for object delivery*, IETF RFC 6330, `https://tools.ietf.org/html/rfc6330`, 2011.

[10] P. Maymounkov, D. Mazières, *Kademlia: A peer-to-peer information system based on the XOR metric*, in *IPTPS '01 revised papers from the First International Workshop on Peer-to-Peer Systems*,

p. 53–65, available at http://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf, 2002.

[11] A. Miller, Yu Xia, et al., *The honey badger of BFT protocols*, Cryptology e-print archive 2016/99, https://eprint.iacr.org/2016/199.pdf, 2016.

[12] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, https://bitcoin.org/bitcoin.pdf, 2008.

[13] S. Peyton Jones, *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*, *Journal of Functional Programming* **2** (2), p. 127–202, 1992.

[14] A. Shokrollahi, M. Luby, *Raptor Codes*, *IEEE Transactions on Information Theory* **6**, no. 3–4 (2006), p. 212–322.

[15] M. van Steen, A. Tanenbaum, *Distributed Systems, 3rd ed.*, 2017.

[16] The Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, Institute for Advanced Study, 2013, available at https://homotopytypetheory.org/book.

[17] G. Wood, *PolkaDot: vision for a heterogeneous multi-chain framework*, draft 1, https://github.com/w3f/polkadot-white-paper/raw/master/PolkaDotPaper.pdf, 2016.

A The TON Coin, or the Gram

The principal cryptocurrency of the TON Blockchain, and in particular of its masterchain and basic workchain, is the *TON Coin*, also known as the *Gram* (GRM). It is used to make deposits required to become a validator; transaction fees, gas payments (i.e., smart-contract message processing fees) and persistent storage payments are also usually collected in Grams.

A.1. Subdivision and terminology. A *Gram* is subdivided into one billion (10^9) smaller units, called *nanograms*, *ngrams* or simply *nanos*. All transfers and account balances are expressed as non-negative integer multiples of nanos. Other units include:

- A *nano*, *ngram* or *nanogram* is the smallest unit, equal to 10^{-9} Grams.
- A *micro* or *microgram* equals one thousand (10^3) nanos.
- A *milli* is one million (10^6) nanos, or one thousandth part (10^{-3}) of a Gram.
- A *Gram* equals one billion (10^9) nanos.
- A *kilogram*, or *kGram*, equals one thousand (10^3) Grams.
- A *megagram*, or *MGram*, equals one million (10^6) Grams, or 10^{15} nanos.
- Finally, a *gigagram*, or *GGram*, equals one billion (10^9) Grams, or 10^{18} nanos.

There will be no need for larger units, because the initial supply of Grams will be limited to five billion ($5 \cdot 10^9$) Grams (i.e., 5 Gigagrams).

A.2. Smaller units for expressing gas prices. If the necessity for smaller units arises, "specks" equal to 2^{-16} nanograms will be used. For example, gas prices may be indicated in specks. However, the actual fee to be paid, computed as the product of the gas price and the amount of gas consumed, will be always rounded down to the nearest multiple of 2^{16} specks and expressed as an integer number of nanos.

A.3. Original supply, mining rewards and inflation. The total supply of Grams is originally limited to 5 Gigagrams (i.e., five billion Grams or $5 \cdot 10^{18}$ nanos).

This supply will increase very slowly, as rewards to validators for mining new masterchain and shardchain blocks accumulate. These rewards would amount to approximately 20% (the exact number may be adjusted in future) of the validator's stake per year, provided the validator diligently performs its duties, signs all blocks, never goes offline and never signs invalid blocks. In this way, the validators will have enough profit to invest into better and faster hardware needed to process the ever growing quantity of users' transactions.

We expect that at most 10%[37] of the total supply of Grams, on average, will be bound in validator stakes at any given moment. This will produce an inflation rate of 2% per year, and as a result, will double the total supply of Grams (to ten Gigagrams) in 35 years. Essentially, this inflation represents a payment made by all members of the community to the validators for keeping the system up and running.

On the other hand, if a validator is caught misbehaving, a part or all of its stake will be taken away as a punishment, and a larger portion of it will subsequently be "burned", decreasing the total supply of Grams. This would lead to deflation. A smaller portion of the fine may be redistributed to the validator or the "fisherman" who committed a proof of the guilty validator's misbehavior.

A.4. Original price of Grams. The price of the first Gram to be sold will equal approximately \$0.1 (USD). Every subsequent Gram to be sold (by the TON Reserve, controlled by the TON Foundation) will be priced one billionth higher than the previous one. In this way, the n-th Gram to be put into circulation will be sold at approximately

$$p(n) \approx 0.1 \cdot (1 + 10^{-9})^n \quad \text{USD}, \tag{26}$$

or an approximately equivalent (because of quickly changing market exchange rates) amount of other (crypto)currencies, such as BTC or ETH.

A.4.1. Exponentially priced cryptocurrencies. We say that the Gram is an *exponentially priced cryptocurrency*, meaning that the price of the n-th

[37]The maximum total amount of validator stakes is a configurable parameter of the blockchain, so this restriction can be enforced by the protocol if necessary.

Gram to be put into circulation is approximately $p(n)$ given by the formula

$$p(n) = p_0 \cdot e^{\alpha n} \tag{27}$$

with specific values $p_0 = 0.1$ USD and $\alpha = 10^{-9}$.

More precisely, a small fraction dn of a new coin is worth $p(n)\,dn$ dollars, once n coins are put into circulation. (Here n is not necessarily an integer.)

Other important parameters of such a cryptocurrency include n, the total number of coins in circulation, and $N \geq n$, the total number of coins that can exist. For the Gram, $N = 5 \cdot 10^9$.

A.4.2. Total price of first n coins. The total price $T(n) = \int_0^n p(n)\,dn \approx p(0) + p(1) + \cdots + p(n-1)$ of the first n coins of an exponentially priced cryptocurrency (e.g., the Gram) to be put into circulation can be computed by

$$T(n) = p_0 \cdot \alpha^{-1}(e^{\alpha n} - 1) \quad . \tag{28}$$

A.4.3. Total price of next Δn coins. The total price $T(n + \Delta n) - T(n)$ of Δn coins put into circulation after n previously existing coins can be computed by

$$T(n+\Delta n) - T(n) = p_0 \cdot \alpha^{-1}(e^{\alpha(n+\Delta n)} - e^{\alpha n}) = p(n) \cdot \alpha^{-1}(e^{\alpha\,\Delta n} - 1) \quad . \tag{29}$$

A.4.4. Buying next coins with total value T. Suppose that n coins have already been put into circulation, and that one wants to spend T (dollars) on buying new coins. The quantity of newly-obtained coins Δn can be computed by putting $T(n + \Delta n) - T(n) = T$ into (29), yielding

$$\Delta n = \alpha^{-1} \log\left(1 + \frac{T \cdot \alpha}{p(n)}\right) \quad . \tag{30}$$

Of course, if $T \lll p(n)\alpha^{-1}$, then $\Delta n \approx T/p(n)$.

A.4.5. Market price of Grams. Of course, if the free market price falls below $p(n) := 0.1 \cdot (1+10^{-9})^n$, once n Grams are put into circulation, nobody would buy new Grams from the TON Reserve; they would choose to buy their Grams on the free market instead, without increasing the total quantity of Grams in circulation. On the other hand, the market price of a Gram cannot become much higher than $p(n)$, otherwise it would make sense to obtain new Grams from the TON Reserve. This means that the market price of Grams

would not be subject to sudden spikes (and drops); this is important because stakes (validator deposits) are frozen for at least one month, and gas prices cannot change too fast either. So, the overall economic stability of the system requires some mechanism that would prevent the exchange rate of the Gram from changing too drastically, such as the one described above.

A.4.6. Buying back the Grams. If the market price of the Gram falls below $0.5 \cdot p(n)$, when there are a total of n Grams in circulation (i.e., not kept on a special account controlled by the TON Reserve), the TON Reserve reserves the right to buy some Grams back and decrease n, the total quantity of Grams in circulation. This may be required to prevent sudden falls of the Gram exchange rate.

A.4.7. Selling new Grams at a higher price. The TON Reserve will sell only up to one half (i.e., $2.5 \cdot 10^9$ Grams) of the total supply of Grams according to the price formula (26). It reserves the right not to sell any of the remaining Grams at all, or to sell them at a higher price than $p(n)$, but never at a lower price (taking into account the uncertainty of quickly changing exchange rates). The rationale here is that once at least half of all Grams have been sold, the total value of the Gram market will be sufficiently high, and it will be more difficult for outside forces to manipulate the exchange rate than it may be at the very beginning of the Gram's deployment.

A.5. Using unallocated Grams. The TON Reserve will use the bulk of "unallocated" Grams (approximately $5 \cdot 10^9 - n$ Grams)—i.e., those residing in the special account of the TON Reserve and some other accounts explicitly linked to it—only as validator stakes (because the TON Foundation itself will likely have to provide most of the validators during the first deployment phase of the TON Blockchain), and for voting in the masterchain for or against proposals concerning changes in the "configurable parameters" and other protocol changes, in the way determined by the TON Foundation (i.e., its creators—the development team). This also means that the TON Foundation will have a majority of votes during the first deployment phase of the TON Blockchain, which may be useful if a lot of parameters end up needing to be adjusted, or if the need arises for hard or soft forks. Later, when less than half of all Grams remain under control of the TON Foundation, the system will become more democratic. Hopefully it will have become more mature by then, without the need to adjust parameters too frequently.

A.5.1. Some unallocated Grams will be given to developers. A predefined (relatively small) quantity of "unallocated" Grams (e.g., 200 Megagrams, equal to 4% of the total supply) will be transferred during the deployment of the TON Blockchain to a special account controlled by the TON Foundation, and then some "rewards" may be paid from this account to the developers of the open source TON software, with a minimum two-year vesting period.

A.5.2. The TON Foundation needs Grams for operational purposes. Recall that the TON Foundation will receive the fiat and cryptocurrency obtained by selling Grams from the TON Reserve, and will use them for the development and deployment of the TON Project. For instance, the original set of validators, as well as an initial set of TON Storage and TON Proxy nodes may be installed by the TON Foundation.

While this is necessary for the quick start of the project, the ultimate goal is to make the project as decentralized as possible. To this end, the TON Foundation may need to encourage installation of third-party validators and TON Storage and TON Proxy nodes—for example, by paying them for storing old blocks of the TON Blockchain or proxying network traffic of a selected subset of services. Such payments will be made in Grams; therefore, the TON Foundation will need a significant amount of Grams for operational purposes.

A.5.3. Taking a pre-arranged amount from the Reserve. The TON Foundation will transfer to its account a small part of the TON Reserve—say, 10% of all coins (i.e. 500 Megagrams) after the end of the initial sale of Grams—to be used for its own purposes as outlined in **A.5.2**. This is best done simultaneously with the transfer of the funds intended for TON developers, as mentioned in **A.5.1**.

After the transfers to the TON Foundation and the TON developers, the TON Reserve price $p(n)$ of the Gram will immediately rise by a certain amount, known in advance. For example, if 10% of all coins are transferred for the purposes of the TON Foundation, and 4% are transferred for the encouragement of the developers, then the total quantity n of coins in circulation will immediately increase by $\Delta n = 7 \cdot 10^8$, with the price of the Gram multiplying by $e^{\alpha\,\Delta n} = e^{0.7} \approx 2$ (i.e, doubling).

The remainding "unallocated" Grams will be used by the TON Reserve as explained above in **A.5**. If the TON Foundation needs any more Grams

thereafter, it will simply convert into Grams some of the funds it had previously obtained during the sale of the coins, either on the free market or by buying Grams from the TON Reserve. To prevent excessive centralization, the TON Foundation will never endeavour to have more than 10% of the total amount of Grams (i.e., 500 Megagrams) on its account.

A.6. Bulk sales of Grams. When a lot of people simultaneously want to buy large amounts of Grams from the TON Reserve, it makes sense not to process their orders immediately, because this would lead to results very dependent on the timing of specific orders and their processing sequence.

Instead, orders for buying Grams may be collected during some predefined period of time (e.g., a day or a month) and then processed all together at once. If k orders with i-th order worth T_i dollars arrive, then the total amount $T = T_1 + T_2 + \cdots + T_k$ is used to buy Δn new coins according to (30), and the sender of the i-th order is allotted $\Delta n \cdot T_i/T$ of these coins. In this way, all buyers obtain their Grams at the same average price of $T/\Delta n$ USD per Gram.

After that, a new round of collecting orders for buying new Grams begins.

When the total value of Gram buying orders becomes low enough, this system of "bulk sales" may be replaced with a system of immediate sales of Grams from the TON Reserve according to formula (30).

The "bulk sales" mechanism will probably be used extensively during the initial phase of collecting investments in the TON Project.