Everscale Whitepaper

By Mitja Goroshevsky

> It is impossible to have a radical philosophy
> without at first sounding like a lunatic or a moron
> — Michael Malice, “The Anarchist Handbook”

Abstract

Bitcoin has eliminated trust when conducting peer-to-peer financial transactions by inventing the economic incentives and technology to run a decentralized computer network. Ethereum has invented Turing complete-programming for it, which helped create more complicated financial instruments, such as tokenized assets, collateralized loans, liquidity pools or synthetic assets. In this paper we describe the architecture, components, governance mechanism and financial model of Everscale blockchain — a trustless worldwide operating system.

Preamble

> We, the undersigned Free TON Community, Validators and Developers, hereby announce the launch of Free TON Blockchain upon the principles and terms stated in this Declaration of Decentralization.
>
> — The Declaration of Decentralization[1]

Everscale is a decentralized global blockchain network launched on top of Ever OS[2] on May 7, 2020. On November 10, 2021 by the decision of its community it was renamed to Everscale from Free TON. Its technology has roots in the TON blockchain, this will be discussed in some more details throughout this paper. Sometimes it was simply impractical to rename everything. For example, the TONIX blockchain file system CLI was not renamed, etc.

Everscale is a new and unique blockchain design that proposes a scalable decentralized world computer, paired with a distributed operating system — Ever OS.

Ever OS is capable of processing millions of transactions per second, with Turing-complete smart contracts and decentralized user interfaces.

Everscale presents some new and unique properties, such as dynamic multithreading, soft majority consensus and distributed programming, which enable it to be scalable, fast and secure at the same time. It is governed by a decentralized community founded upon meritocratic principles via Soft Majority Voting protocol.

Everscale has powerful developer tools, such as compilers for Solidity and C++, sdk and api, client libraries ported to more than 20 languages and platforms, a range of decentralized browsers and wallets empowering many applications in DeFi, NFT, tokenization and governance domains.

[1] https://freeton.org/dod
[2] https://everos.dev

Chapter One
Everscale

Decentralization

1. Cryptocurrency provides economic incentives to the decentralized computer we call blockchain and is inseparable from the technology that blockchain is empowered with. Everyone who says otherwise is just protecting outdated values for their own benefit. The reason for this inseparability is the “decentralization” property of the system. The whole raison d'être of distributed computer technology is its ability to be run by parties which do not need to trust each other. Please note that the emphasis is not on “do not *trust*” but on “do not *need to trust.*” This is an important and powerful concept. There is already technology which can run distributed systems. These systems can scale, they have sharding, consensus, advanced databases, programming languages, they are proven by many years in production to deliver robust performance. There is just one little thing, one detail, one tiny principle you need to obey in order to use them: you must trust someone who runs them.
2. That trust assumption is how we, as a human race, surrendered our freedom to a few tech corporations and a handful of government organisations around the world. This trust assumption is now almost unilaterally accepted by people as something mandatory to protect them, to provide security guarantees, economy of scale, easy to use interfaces, usability, utility and so on. Let’s call them the conveniences of surrender. The evil gifts.
3. The inability as a group to protect ourselves which leads to a delegation of power is a compromise we make with our liberal values. As human beings following the laws of history, any time and every time we have a technical ability to end such compromise — we must.
4. Mr. Wilhelm Steinitz, the first chess world champion and the founding father of chess theory, has formulated one of the important strategic principles — if a player gains a positional advantage over another, it must attack to win[3]. Our life is no chess game, so the price we pay for losing could be catastrophic.
5. One of the founding fathers of the United States of America and its second president John Adams, stated: “Our Consolation must be this, my dear, that Cities may be rebuilt, and a People reduced to Poverty, may acquire fresh Property: But a Constitution of Government once changed from Freedom, can never be restored. Liberty once lost is lost forever. When the People once surrender their share in the Legislature, and their Right of defending the

[3] Lasker's Manual of Chess by Emanuel Lasker, Russel Enterprises, Inc, Milford, CT, USA 2008

Limitations upon the Government, and of resisting every Encroachment upon them, they can never regain it."[4]

6. Or if one prefers George Orwell's shorter version: "We know that no one ever seizes power with the intention of relinquishing it**."**[5] Over 70 years later, we couldn't be any closer to 1984.
7. Fortunately, Satoshi Nakamoto invented the economic incentives and technology for running a decentralized computer network. Such a supposedly simple idea gave human individuals an advantage in the game against a human collective. Ethereum has invented turing complete-programming for it and Everscale has invented its operating system. The Operating System of Freedom, that is. The tool to fight a Borg within us[6], the conveniences of surrender, the evil gifts.

Meritocratic Token Distribution

> "Either products do actually exchange in the long run in proportion to the labor attaching to them—in which case an equalization of the gains of capital is impossible; or there is an equalization of the gains of capital—in which case it is impossible that products should continue to exchange in proportion to the labor attaching to them."[7]

8. One of the most advanced arguments against the trustless system is that such a system, as decentralized as it may seem, is only so on a level of its protocols. Such a network will presumably scale back into centralization when needed to reach certain decisions for continuous operations requiring social consensus. That once decision making is involved, it necessarily will lead to centralization if the system wants to be somewhat efficient. And despite the efforts of the crypto community to bring decentralization into the protocol governance itself there are still plenty of examples where this argument prevails. Bitcoin is still keeping its software code base in a centralized repository. Very few Ethereum and Bitcoin mining pools control most of the mining power of both networks. The blockchain foundations are everywhere, centrally running core development of their protocols and distributing grants to developers teams. Most of the newer proof-of-stake networks pre-sold their tokens to a few large venture investors and now control their grant system and protocol development. In general most of the leading blockchains tokens are controlled by a very few investors.
9. We will discuss how Everscale tackles all those issues throughout this paper. Let's start from the most obvious one, how the tokens are distributed in the first place.

---

[4] John Adams to Abigail Adams, 7 July 1775

[5] George Orwell "1984"

[6] https://en.wikipedia.org/wiki/Borg

[7] Eugen von Böhm-Bawerk, "Capital and Interest", 1st edition, p. 362

10. One of the characteristics of proof-of-stake (POS) design is that it requires validators to have a material stake in the network which they would be afraid of losing. This assumption provides basic game-theory ground behind POS. Participants are motivated to ensure the correctness of the blockchain by the possibility of losing their stakes if they don't.
11. Usually POS blockchains begin with selling their tokens to future validators to create a starting point of this game economy. In Everscale it was very clear to everybody from the very beginning that we would never sell tokens to anybody. The puzzle that we had to solve was how to distribute the tokens in a way permitted by the game theory of POS.
12. I believe we found a revolutionary solution to that problem. It's something we call the Meritocratic Token Distribution (MTD) model. It starts with the community member proposing a Contest in which all other members of the community may participate. The contest is discussed and if agreed that the end result of this Contest will benefit the community and the network as a whole, the Rewards for it is voted for, the Jury is selected, the participants are providing solutions according to the contest specs, the Jury is voting for the solutions, and tokens are distributed to the winners. All those activities are concluded on-chain.
13. Pareto optimality[8] is achieved as follows: token holders are distributing tokens for the perceived increase of the value of the network. This is obtained through the efforts of different active participants from decentralized communities which exchange their labor for native network token, thus turning themselves into token holders. Tokens provide an opportunity to participate in all projects that are being run on Everscale by representing a share of their governance. The Byzantine Fault Tolerance **Governance** (BFTG)[9] is an ever-evolving set of rules implemented as smart contracts, ensuring that the system is not gamed by any kind of collusion between any kind of parties.
14. So, to summarize, instead of minting tokens by burning electricity or for production of empty blocks, the tokens are distributed meritocratically. Of course, Everscale is paying for empty block production as well, but the mechanism of such payment is quite different, as will be discussed below.
15. Yet even with probably the most advanced and decentralized method of token distribution, the question remains that if for a right market price (as high as it might be) the tokens will eventually end up with the same crypto investor, what difference does it make how they were distributed in the first place?
16. The difference between Meritocratic Token Distribution and public or private placement of those tokens is attributed to two facts: a) when tokens are distributed meritocratically it ends up in the hands of different entrepreneurs building the network ecosystem, not just in the hands of core protocol developers, resembling the deployment of venture capital. Needless to

[8] https://en.wikipedia.org/wiki/Pareto_efficiency

[9] Practical BFT Governance

say, this is what empowers the economy; and b) it has a by-product: governance tokens[10] of the project the entrepreneurs had created during the contest they participated in.

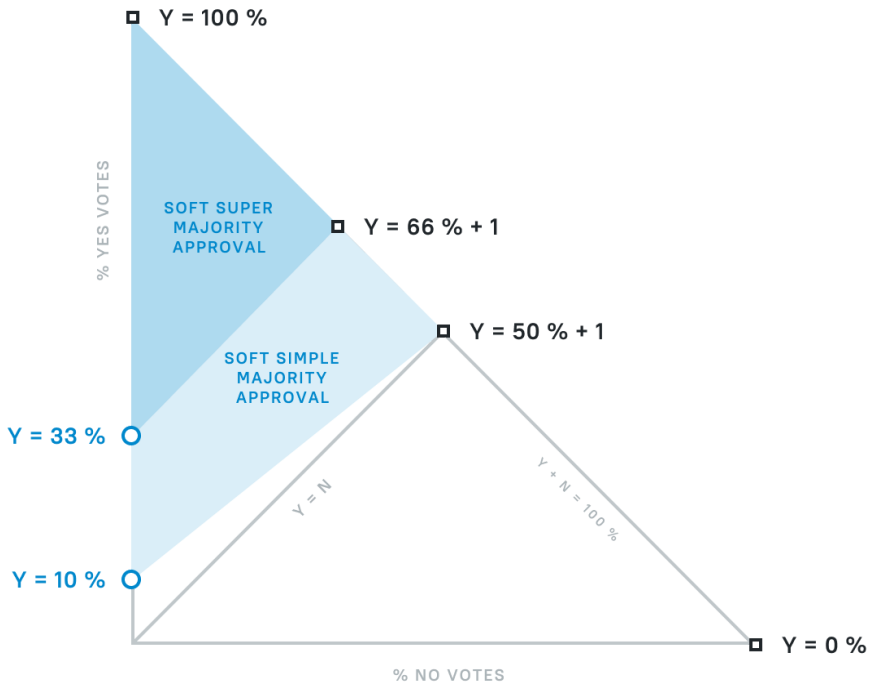
17. If entrepreneurs are in it for the long run with the project they created, they will need tokens to pay for gas and their bills. These tokens will end up on the market. Here usually lies the disconnect between the projects that were partially created by the meritocratic process and the usual token distribution. The value of that token is created only by the utility of that decentralized platform through the gas payment, unless the token itself also becomes a governance token for all the projects supported by it.
18. Let's imagine a community Giver that, as part of the MTD process, will ask contest participants to create a governance token for their contest submission and submit some portion of that token under that Giver's control in exchange for a contest winning payment. Now we not only have the Meritocratic Token Distribution mechanism but Decentralized Participation as well.
19. For example, a Decentralized Name Service (DeNS) is used as a basic filename and directory structure in the Ever OS file system. The rules of this governance token may allow those who hold it to execute certain rights regarding the project. By itself, they may or may not have any monetary value. Nevertheless, usually such tokens will give certain monetary rights in the project, such as the case with DeNS where they could be exchanged or burned against for a certain amount of EVERs collected by the project as payment for registrations of DeNames.
20. Now, if a governance token provides certain decision power in the project but can be exchanged or burned against the project's collected fees, we have a situation where a holder can not receive accrued fees until and unless she exits the project. The exit price may be automatically calculated as total native tokens collected divided by the total amount of governance tokens outstanding multiplied by the amount of tokens to be exchanged. Of course apart from exiting the project, the holder could trade such tokens with a premium on one of the Everscale decentralized exchanges It may be wise to add a mechanism which would mint such governance tokens as a declining function (similar to halving) versus a fixed rate. Additionally, a mechanism could be imagined where users are able to commit funds directly to the project account to mint such tokens i.e. initial decentralized offering (IDO).
21. Let’s not discuss here the price discovery of any of those trades. What is important is that the Everscale native crypto currency is used not only as a store of value by the scarcity of its reserves but also as means to participate in all projects created directly as an outcome of the Meritocratic Token Distribution mechanism.
22. In sum, people who perform work will usually want to receive a perceived “stable coin” for the time and effort they spent, because they can easily spend these tokens for consumption. Of course, the “Stable Coin” name is a deception. In fact, we would rather call that kind of token a “DeValue” token because, as any fiat currency, it will, and should, gradually lose its value to be an attractive medium of exchange. The “Capitalist” is someone who prefers to hold onto a token which “stores” value. This is also deceiving because in order to be attractive this token

[10] Governance token is a token that gives its owner some decision making rights in the project.

needs to gradually increase its value over time in comparison with prices of goods and services. We call that token a Native Token. The entrepreneur is someone who creates value. The Native Token therefore is capturing not only the platform fees paid in gas, not only the general demand for a scarce resource, but also part of the value created by all entrepreneurs' projects which received tokens through the MTD process.

Governance

23. Everscale governance is very simple — anyone can upload a proposal for which the community votes with their tokens using the Soft Majority Voting (SMV) mechanism.
24. Soft majority voting is a transparent voting process with advance announcement and clear deadline. If members don't have an opinion and/or are reluctant to vote, it is assumed that they are neutral. Instead of making an attempt to force everyone to vote for every decision, SMV essentially allows decisions by those who care.
25. The metric for passing a decision is the difference between % of Yes minus % of No. For example, if 10% of voters said Yes and none said No, then it is assumed the decision is sufficiently supported and there are no objections. At the same time, if all members voted, then a simple majority rule applies: 50%+1 vote for passing a decision. When we connect those two dots on a graph with % of Yes and % of No axes, we get a "soft" simple majority threshold line.
26. For important decisions like amendment of constitutional community documents, we can draw a "soft" super-majority threshold line (see Diagram below). Soft majority voting mechanism is programmed on Everscale via SMV Smart Contract.



27. In his work “Moving beyond coin voting governance” Vitalik Buterin identifies vote buying as a major threat to the decentralized on-chain coin voting governance model. It describes at length a possibility of an attack on a governance system by an automated smart contract that auctions voting rights in exchange for some tokens. The SMV protocol is a unique voting mechanism proposed by Pavel Prigolovko, optimized for low participation. It is not a simple token holders voting solution. In fact, by addressing a low participation problem it also has other implications. Remember that in SMV each negative vote increases the decision passing threshold. Therefore the attacker will run into ever increasing cost of vote buying because the honest participants' negative votes have more voting power. In an important decision the attacker may need to buy 75% or more of all tokens for the decision to pass. In effect this is a form of quadratic voting already.
28. As mentioned above, it is quite a simple system; yet taken in conjunction with the Meritocratic Token Distribution, it becomes a very powerful tool for solving many governance problems.

Economy

29. Our target is to create an economy for the operating system whose main distinguishing asset is its malleability.
30. As some of you may know Nasim Taleb did not invent antifragile[11]. It was Andrei Tarkovsky’s invention, as he put it in “Stalker”: "Weakness is a great thing, and strength is nothing. When a man is just born, he is weak and flexible. When he dies, he is hard and insensitive. When a tree is growing, it's tender and pliant. But when it's dry and hard, it dies. Hardness and strength are death's companions. Pliancy and weakness are expressions of the freshness of being. Because what has hardened will never win.” We believe “malleability” is a good term to replace “antifragile”.
31. If the Governance token is a unit of account of certain rights within a particular decentralized project then the economic system becomes malleable. Of course, if the Governance token loses its value all the time, less people will be inclined to hold such a token. Therefore, the token itself should be a store of value, which contradicts its other use - as money.
32. The reasons for that are discussed in “EVER and NEVER Binary system” paper[12]:
33. “According to the current economic teachings, money has three main characteristics: medium of exchange, unit of account, and store of value. This definition may have been correct 100 years ago. Today it is nothing but a lie[13].”

---

[11] https://en.wikipedia.org/wiki/Antifragile

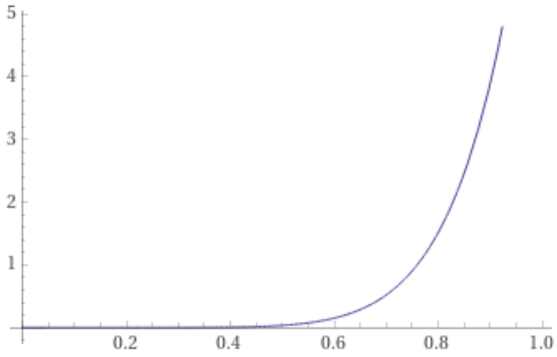
[12] EVER and NEVER Binary System

[13] Mankiw, N. Gregory (2007). "2". *Macroeconomics* (6th ed.). New York: Worth Publishers. pp. 22–32. ISBN 978-0-7167-6213-3.

34. “In fact it would probably be correct to say that the money in the modern economy must gradually lose its value in order to be an attractive medium of exchange. Quite simply when a person holds on to something that loses its value over time, it will most likely try to exchange it with something more valuable. Such a person would not hesitate going to a shop and buying not only things they dearly need, such as pizza, but also things they do not need so much, such as entertainment, or things they don’t need at all, such as a new phone.”
35. From a pure practical perspective, therefore, if EVER is a value capturing token, another native token should be created as an integral part of Everscale that would be used for money transactions and that would lose value while EVER would gain. As mentioned above such a token is NEVER. Some part of the EVER supply should be used as a NEVER stability fund as proposed in the “EVER and NEVER Binary system” paper and as will be described below.
36. Since EVER token is not money, it should not be used for network usage fee payments, or such fees will constantly diminish. In order to achieve that, the dynamic gas price should be introduced, but unlike in other blockchains, it should not be set forward by network validators. In general, letting validators decide on transaction inclusion, order and price, has been one of the worst ideas of our time.
37. Yet, the price for gas should not be too low, or better, should not be arbitrarily low, by reason of spam prevention. Imagine: if the network fees are too low and the network has an upward capacity limit, it is quite simple to calculate how much money one would need to pay to buy out 100% of such network capacity for X amount of time in the form of shilling (or penny) attack. If the price is very low, the network will constantly be out of gas.
38. Thus, let’s have a dynamic network price function derived from the number of transactions as a percentage of current perceived network capacity and some starting gas price which could be calculated from a perceived cost of running a validator node necessary to run Ever OS, multiplied by the number of such validators necessary to run current Ever Kernel configuration. As will be discussed later, Everscale is almost infinitely scalable, which makes things easier in terms of gas payments, but still in order for it to scale more validators need to join, therefore gas prices should reflect the growing need to attract new validators. The transaction price should start from almost zero and once it reaches the capacity of a Processing WorkChain (see WorkChains) the total price should accommodate the launch of a new WorkChain with a minimum set of Validators. Moreover the premium of the gas price increase should not be paid to current validators, but should be returned to the Validator Giver in order to accommodate the start of a new workchain.

    Let X-axis be the workchain capacity fullness, let Y-axis be the gas price.
    Let the initial price “a” be very close to 0.
    Let's choose a simple power function so that the area under the graph is equal to 1.
    Let f(x) = a + b * x^n
    Then the integral of this function from 0 to 1 is equal to a + b / (n + 1) = 1
    Neglecting the small value of "a" can put b = n + 1

Then f(x) = a+x^n*(n+1)



39. The higher the degree, the more “sharp” the function. At point 1, the function takes on the value (a + b) ~ b
40. Another aspect that needs to be discussed is a payment for empty blocks. After all, without such payments invented by Bitcoin, there would be no cryptocurrency. In the light of this, we believe there should be no emission of new EVER.
41. All EVER s were organized at launch into three accounts we call “Givers”: Validator, Developer and Governance Givers[14]. The tokens from these Givers are distributed through the MTD mechanism by the community.
42. The Validator Giver should pay for every block validators produce (1T for shardblock and 1.7T for Masterblock, changeable only by the community decision if necessary). The total supply of tokens should be kept to around 5.04 bln EVERs. No new tokens should ever be issued. When Validator Giver runs out of tokens, presumably there will be no empty blocks.
43. Following the malleability principle, the economic system of Everscale should accommodate for other forms of token distribution or public funding as the community would see fit from time to time. The new givers would be formed, funds would go to support experiments in funding models. It is our firm belief that the guiding principle to all these experiments, though, should be that no funds are distributed to a project without an entrepreneur behind it having vested interest in it and risk associated with it.

###

44. While trustless gold can decentralize central banks and trustless applications can decentralize finance, only a trustless operating system can decentralize the economy and get rid of monopoly altogether.
45. By monopoly we mean control over any market by a single entity. In economic theory the monopoly is always referred to in a negative connotation because it destroys the open market

[14] It is currently called “Referral Giver” which we believe needs another name, we suggest “Governance Giver” instead.

competition, which, in turn, affects the price and motivation system behind free market economy thus greatly diminishing progress. In most modern economies monopolies are prohibited by law. Yet, the fight against them is spotty and ineffective as almost everything governments usually do precisely because any government is a monopoly by itself. Sometimes they execute monopoly power over just a few aspects such as the use of force, and sometimes they monopolize almost everything. Therefore let's assume there is no difference between government monopoly and enterprise monopoly, as they all are equally bad. Bottom line is, there is no effective mechanism to fight against monopoly in a modern society.

46. Cryptocurrency can solve that problem. First of all, it is almost impossible to purchase another project in a decentralized world because of a potential fork[15]. The power of decentralized applications runs in the immutability of its system software, yet at the same time in the free and open nature of its source code. Governance tokens, unlike shares in a company, give very little control over the project software, management or operations. Such decisions as projects merge can be done in a friendly manner by simply transferring the talent from one team to another, but nobody in this case would prevent a project from being picked up by another team, continued by its community or simply forked. That changes an economic reality completely, rendering all asset holders protection agencies unnecessary in the future when blockchain technology and cryptocurrency will dominantly empower the world economy.
47. Technically, though, in order to achieve that future, a design should propose a truly decentralized, distributed, secure, sustainable, scalable, low latency world computer, together with an operating system.
48. This paper discusses the design of such a system from three perspectives: technical, governance and economic. Because of the sheer volume of the topics to cover, it should be viewed as a framework of ideas and design blueprints to be further discussed in separate documents, some of which are already available (links to them will be provided throughout the paper), and some are still under development.

[15] https://cointelegraph.com/news/hive-hard-fork-is-successful-steem-crashes-back-to-earth

Chapter Two
Ever Kernel (EK)

Context

49. In 2018 Dr. Nikolai Durov[16] released a series of papers in which he described a Virtual Machine, network and consensus protocols, which were dubbed TON. In late 2019 a prototype of C++ network node implementation was released by Dr. Durov and his team. Starting from 2018, TON Labs, using just documents available at that date, started to work on a parallel and completely independent implementation of the protocol in Rust programming language. In the first half of 2020 the full node implementation in Rust was released followed by the validator node release in late 2020.
50. In computer system terms we view Dr. Durov's design as a distributed virtual microprocessor or network operating system kernel, thus it is called Ever Kernel or EK version 1.0 or similar, throughout that paper. While the design laid a very solid foundation, to answer the requirements of a truly decentralized, distributed, secure, sustainable, scalable, low latency world computer, it had to be developed further. When TON Labs created Ever Operating System and community has launched the Everscale network, the design and real usage requirements of the network demanded not only additional functionality and modules on top of the Kernel, but quite significant changes in the underlying architecture itself, which is often the case in any computing system design evolution (let's call it Ever OS Kernel).
51. The Ever Kernel provides a system with the following components: virtual processing engine (Virtual Machine), network protocol stack responsible for addressing, virtual messaging buses connecting Virtual Machines (DHT, ADNL, Overlay, Broadcast and RLDP protocols), and a consensus layer that synchronizes and validates the execution of smart contract code on Virtual Machines of different physical computers over the network (Catchain, BFT).
52. To that stack of EK protocols we have made several additions that will be discussed in some details later in this section:

    REMP (Reliable External Messaging Protocol)
    MBPP (MasterChain Block Propagation Protocol)
    SMFT (Soft Majority Fault Tolerance)
    MSRP (Masterchain Slashing and Recovery Protocol)
    DDVS (Distributed Dynamic Validator Set)

[16] https://en.wikipedia.org/wiki/Nikolai_Durov

53. Many enhancements and optimizations are constantly made to the existing EK protocols , but they are largely falling outside of the scope of this document. It is important to mention though, that some of these changes are incompatible with the first version of Ever Kernel. We will concentrate in this paper just on the significant changes we made, leaving the smaller patches and changes out. We will discuss the code handling, bootstrapping and upgradability issues in a separate chapter.

WorkChains

54. WorkChains in Ever Kernel v1.0 are not supported apart from the basic (0) workchain while they are fully supported in the Ever OS Kernel. Many rules and messaging protocols to support direct interaction between Workchains were added.
55. One should remember that all Validators of a WorkChain store all the data of that WorkChain exchanging all the blocks produced within such a Workchain and making it a Shard in a strict database design definition.
56. WorkChains could be viewed as distributed processor cores or even as distributed peripheral functionality. They can have different functionality, but they share the same block and consensus structures. Some WorkChains will simply scale the number of accounts out of the practical limitation of a single WorkChain to process data related to a certain number of addresses (let's call them "processing WorkChains"). Others will support additional functionality, such as DriveChain for persistent storage, or IceChain for long term archives as will be explained in some details later, and so on (let's call them "peripheral WorkChains").
57. Processing WorkChains are created dynamically at the time of the elections if the total capacity of all current processing WorkChains are utilized by about 90%, and some other parameters are met. When added, the Processing WorkChain starts operating once validators from a dynamic validator set are allocated to this workchain and synchronized. The address space of such a WorkChain is a division of the address space of the latest available Processing WorkChain, similar to the thread address split/merge mechanism.
58. Peripheral WorkChains are added by the decision of the Network Governance.
59. The validator set with the same capabilities is assigned to a new WorkChain during the elections (or, more precisely, during the D'Elector contract execution, see below).
60. Other types of WorkChains could be theoretically imagined, but the rules of a block commitment into MasterChain should be followed. Theoretically, one can also imagine that another blockchain could be added to Ever OS as a workchain if validators of that blockchain decide to convert the block into a necessary structure and post capabilities required for such a blockchain. In effect, it will turn such a WorkChain into a decentralized bridge to that other network. For instance, to add Ethereum 2.0 network as Everscale WorkChain, the stated capabilities should include a certain Ethereum client for EVM and ETH 2.0 protocol processing,

the ability to wrap Ethereum blocks into Everscale block structure and proof that such Validator is also a validator in Ethereum. Those WorkChain validators will of course deposit their stakes into the D'Elector contracts and produce the same Everscale consensus guarantees for the Ethereum network while completely disregarding that network's own security assumptions, making it in a way even more secure (as both would run in parallel).

61. There are many ways to submit such transactions on the Ethereum side. For example this workchain validators could form some Layer 2 solution for Ethereum, designs of which are discussed in the Ethereum community at length and are not in the scope of this paper.
62. Gas would be paid for transactions within such WorkChain in Everscale native currency, while EVM transactions would be paid in Ether. Yet, nothing may stop these WorkChain validators from supporting transactions within their WorkChain, which will, for example, execute a conversion of Ethereum native currency to Everscale native currency, or vice versa, provide proof of blocks, transactions etc. The number of validators and their stakes will therefore guarantee that WorkChain is correct.
63. Another difference between Processing and Peripheral Workchains is their address space. The peripheral WorkChains may have special address prefixes while Processing Workchains share the same address space.
64. One of the hardest problems in the multi-sharded approach has been inter-shard communications. If our system is heterogeneous, then we can assume it may not need any specific guarantees with respect to the messages that one such shard sends to another. Yet, it almost certainly creates quite an awful user and developer experience. That would be like adding a horizontal layered approach to the vertical layered approach discussed above. The more layers in which a developer application resides , the more difficult it is to use.
65. Of course, it is immensely more difficult to design a homogeneous system. In a truly homogeneous system we want to abstract interaction of different system components between each other and the user. So for a smart contract in one thread of a particular WorkChain, it won’t make any difference sending messages to another smart contract, no matter which thread or WorkChain it resides in.
66. There are two situations when a contract is sending a message to another contract: (i) when the contract knows the exact address of such contract and (ii) when it does not, in which case it will calculate the address (see “Distributed Programming”).
67. When a contract sends a message to a Peripheral Workchain, it must know the destination address and a prefix of such a WorkChain, which is as logical as sending a document from an editing program to a printer for instance.
68. When a contract emits an internal message to another address, and if this address is in another WorkChain, the internal message will be converted into REMP message and sent externally (see REMP below). Such messages are sent to the corresponding WorkChain’s thread validators and placed in a "message queue catchain" (MQC), which is part of the REMP protocol. But in addition to regular external messages, if the external message originated in another WorkChain, it has a special status of “internal-external” message (IEM), to which a

proof of such a message being part of the particular WorkChain block is added. Such messages are then added to the destination's WorkChain thread block just as any internal message of that WorkChain would be.

Multithreading

> "*A thread is a unit of execution on concurrent programming. Multithreading is a technique which allows a CPU to execute many tasks of one process at the same time. These threads can execute individually while sharing their resources.*"[17]

69. Multithreading in the Ever OS Kernel allows for parallel execution of smart contracts by subgroups of the validator set that share the same data. It allows for fast validator rotation, execution parallelism and other features not covered in the Ever Kernel 1.0.
70. Multithreading parallel execution enables each shard in a EK to scale to execution capacity levels only constrained by node network connections and interfaces.
71. Validators within a WorkChain are all divided into rotating groups, each of such groups serving only a subset of smart contracts, something we call "threads", dividing accounts' address space by their number. Number of threads is a configuration parameter of such WorkChain, which can be adjusted depending on a network load to exchange all of the blocks of all of the threads. With a global increase of network throughput, the number of threads could increase.
72. Our WorkChain has a practical limit of about 256 threads currently, because of network load to exchange all blocks of all threads. With a global increase of network throughput, the number of threads could increase.
73. Since all threads share the same data, it is incorrect to call them "shardchains"; the term "thread" should be used instead. Also because of that same reason, there is no need for hypercube routing of the messages between threads as they only add unnecessary delays and produce many more messages.

SMFT (Soft Majority Fault Tolerance) Consensus

74. EK has a BFT-based consensus protocol. It has 4 phases (approve, vote, pre-commit and commit) in order to ensure the consensus is reached. Basic assumptions of BFT consensus is that at least ⅔ of all network validators are honest. It means that a collusion of more than ⅓ of validators can result in a network halt. If more than ⅔ of validators are colluded then network corruption can occur.

[17] https://www.guru99.com/cpu-core-multicore-thread.html#4

75. When a validator detects a non-valid block, it broadcasts a Reject message.
76. One of the problems of any BFT consensus-based network, is that in order to reach a consensus, it has to pass all the protocol phases, which results in network delays because of the number of messages the nodes need to exchange. In the case of Catchain it is `n*log(n)` messages where `n` is the number of nodes in the network.
77. Thus, any BFT based consensus, while assuming that a communication could be malicious at any given time, is also assuming some large percentage of its participants is honest.
78. First of all, the assumption that the majority of participants in a social construct are honest, does not necessarily mean that they are right. A lot of times it is a minority that is both honest and correct. The more complicated the subject of a decision to be made is, the more chances there will be groups with different opinions. Therefore, the network Validators consensus should remain in a very simple and limited capacity, and mostly related to technical aspects of network operations. Validators' own stakes should be subject to slashing by non-validators' stakes (for more on that see DePools).
79. Second, why is this assumption made at all? What exactly stimulates the majority of network nodes to be honest in the first place? There must be some motivation of network participants to behave one way or another. This motivation lies outside of any particular protocol and relates to the economics of that protocol.
80. In the Proof-of-Stake network the validator's motivation to approve correct blocks is secured by their fear of losing their stake if they do not. There is no sound ground to believe that a particular percentage of validators should be honest, and why, apart from game theory arguments.
81. “Note that there have been some recent attempts to develop consensus algorithms based on traditional Byzantine fault tolerance theory[18]; however, all such approaches are based on an M-of-N security model, and the concept of “Byzantine fault tolerance” by itself still leaves open the question of which set the N should be sampled from. In most cases, the set used is stakeholders, so we will treat such neo-BFT paradigms as simply being clever subcategories of “proof of stake”.” [19]
82. Therefore, the whole point of viewing BFT consensus having, say, ⅔ of honest nodes as theoretically secure is misleading at best. And game theory arguments are weak, because they can not prove anything outside of a closed system and to view a blockchain as a closed game system is naive.
83. Let us assume that there are two coins and a market for them with derivatives and margin trading (of course there are many such markets in existence). It is quite easy to prove that holding a large stake in one network and being able to short another may provide a better incentive which will destroy the security motivation of the latter network validators. Large stake holder may short the other network token using his long position as collateral with margin and

[18] http://en.wikipedia.org/wiki/Byzantine_fault_tolerance

[19] “Proof of Stake: How I Learned to Love Weak Subjectivity” by Vitalik Buterin

pay say half of the premium to any validators who will prove to destroy the second network value. It will be enough to have ⅓ of the stakes of the second network to stop it completely and destroy its token value. Having a large number of network participants helps only in the sense of communication and coordination hurdle such attacks could carry, but as recent examples of coordinate stock market pumps shows[20] — this is by far not a security guarantee.

84. The problem multiplies when the network is partitioned. If there is a subset of validators which could corrupt a thread, it will, by proxy, reflect on the security of the whole network.
85. As a general rule if a simple majority of token holders wants the network to continue — nobody should be able to prevent that. It is not currently so in any proof-of-stake networks.
86. In a BFT consensus algorithm, network participants agree upon a block. In the case of a single chain of blocks (no sharding, no threading) this block agreement is final, unless a fisherman detects a problem in it and blames validators for dishonesty. Remember that fishermen are not a part of the consensus. In fact we can say that fisherman is a representative of all other network participants who are not part of the validator consensus, but who want to secure the network.
87. A fisherman is someone who is motivated to secure the network by a potential reward. Fishermen do not need to be a token holder. Of course fishermen can not guarantee anything because there is no guarantee that fishermen will do anything. There are many more problems with fishermen we discuss in "Practical Byzantine Dynamic Slashing (PBDS)"[21]. For the purpose of this discussion what is interesting is that the security of the network relies on a party which does not participate in its "consensus".
88. The notion of Time is critical. It is safe to assume that a block approved and included in the chain even by a single node 10 years ago would most probably be correct, or not, but if nobody cares why is it important? Therefore, the block validity is a function not only of a number of consensus participants but of time. There is simply a threshold function of time/consensus participants that other network participants are ready to accept as an agreement on block finality.
89. This is similar to Bitcoin transaction subjective finality, where a participant will wait for some number of blocks to be mined on top of the block in question for it to ensure its validity (since each mined block increases the hash power spent, it may be prohibitively expensive to reverse the transactions after a certain number of blocks (usually 6)).[22]
90. What is the problem with such an approach? The finality time. We need to wait 6 Bitcoin blocks (about an hour) or 32 Solana slots (about 13 sec.) for a transaction to be probabilistically secured.

---

[20] https://www.cnbc.com/2021/01/28/gamestop-now-called-a-pump-and-dump-scheme-what-you-need-to-know.html

[21] https://forum.freeton.org/t/practical-byzantine-dynamic-slashing-pbds/2519

[22] In a proof-of-stake, this idea is used, for example, by Solana Tower BFT: https://docs.solana.com/implemented-proposals/tower-bft

91. The BFT finality is faster since it takes about ~5 sec. to agree on a block by ⅔ of 100 nodes. But is 100 nodes enough to be considered a decentralized network and is even 5 sec. acceptable?
92. Theoretically we would like to have a consensus protocol that can produce blocks as fast as 500 ms. and have a sub-second finality, with the security guarantees provided by a large set of nodes, say north of 10,000 assuming just ½+1 are honest. Is that even possible?
93. Remember our general assumption that most of the time most of the validators are honest. Indeed it is hard to imagine a network which would constantly pay fishermen success fees. Such a network would halt after several blocks. Occasionally there are bad actors which believe they would defy the protocol and try to corrupt it. But all current protocols all of the time are optimised for the event that is extremely rare.
94. Let's make a protocol that would come to a consensus using the Soft Majority Voting mechanism with some modifications. Remember that in soft majority voting the consensus is reached after some small number of positive Votes, say 20%, while no single negative vote is received. If at least one negative vote is received, the consensus threshold is increased until the point where the majority threshold is set (51% for simple majority).
95. Of course this will lead to the following problems:
    a. If a collator collides with just 20% of the nodes, it will not transmit blocks to other validators of the set, quickly approve the block and send it to masterchain without even a single node being able to send a Reject.
    b. The same malicious set can create a message from thin air and write it into that rogue block. The message will be viewed as valid by another shard that will execute it within another smart contract of that shard.
    c. Because Everscale is very fast, all those operations will happen in a few seconds time frame (masterblock validation ~4 sec, shard block creation ~3 sec) and that time won't be enough to stop the attack.
    d. In a paradigm of distributed programming, when the smart contract is an object which interacts with other objects asynchronously the chain of interactions between those objects could be very long. Since it is impossible to predict what messages smart contracts will emit before executing them it will be very hard to correct any corruption in a block in the middle of such a transaction chain.
    e. Once executed, such a transaction could allow attackers to exit with large sums of money using decentralized exchanges and bridges with other networks which could render all the staking security guarantees useless.
    f. In which event the fishermen will be too late for the party. The fork of a blockchain (whether in a form of vertical block or just fork) will recover a very insignificant portion of the attacker's reward. The blockchain will suffer catastrophic perception damage.

96. Let's emphasize that the idea that we could correct such an action retrospectively is false. Forks, vertical blocks or any other such mechanism won't be able to mitigate such a catastrophic event. Therefore the whole security assumption relying on fishermen is wrong.
97. Let also emphasize that guaranteeing a security of a native token in a shard does not provide enough security guarantees as it should also guarantee the smart contract execution correctness on that shard, facing catastrophic consequences if not.

Let us present a Soft Majority Fault Tolerance protocol (or SMFT for short):[23]

98. Will leave the Catchain BFT protocol intact as of the leadership selection and consensus phazes. The Thread validators will validate the block as they are now, just with the BFT threshold lowered to say 51% signatures, which will be used for preventing natural forking and slashing in case of an attack.[24]
99. In addition, let's have a protocol that samples some random nodes out of the WorkChain Validator set and requires them to validate every block (let's call them "Verifiers"). The randomness must be calculated after the block candidate has been propagated. The protocol must be non-interactive, succinct and low latency.
100. Let us concentrate on the block collation phaze, before the consensus, because in order for malicious validators to validate the malicious block it should be collated in the first place. If we prove that the malicious collation will be impractical, it won't matter how many potentially malicious validators the thread has.
101. Instead of broadcasting the block that has been already validated, we will ask collator to broadcast the block candidate to all nodes in the WorkChain. If a collator won't do this, the block header won't be included into Masterchain and the Collator will be slashed for data withholding.
102. Before we describe how blocks are verified let's prove that the block has indeed been broadcast. Without such proof the malicious set of actors could siphon the block only to themselves, "mining" the necessary outcome.
103. Let some validators be defined as Broadcast Protectors (BPs for short).
104. When receiving a new block broadcast from the collator, the validators calculate a set of BPs from a hash of the block and list of validators. Say, taking the Workchain Validator Set divided by some number.
105. Validators will send a Block Receipt — a block hash signed by a BLS signature to all BPs. When a BP collects 51% of Receipts it will broadcast a multiplied Receipts and a numbered list

[23] SMFT was developed by Mitja Goroshevsky and Pavel Prigolovko with assistance from Kirill Zavorovsky, Dmitry Shtukeberg, Leonid Kholodov (all — TON Labs) and Andrey Lyashin (Pruvendo). The full description of the protocol, detailed proofs and math model is published separately.

[24] In theory we can throw away the BFT protocol altogether, but further research is needed.

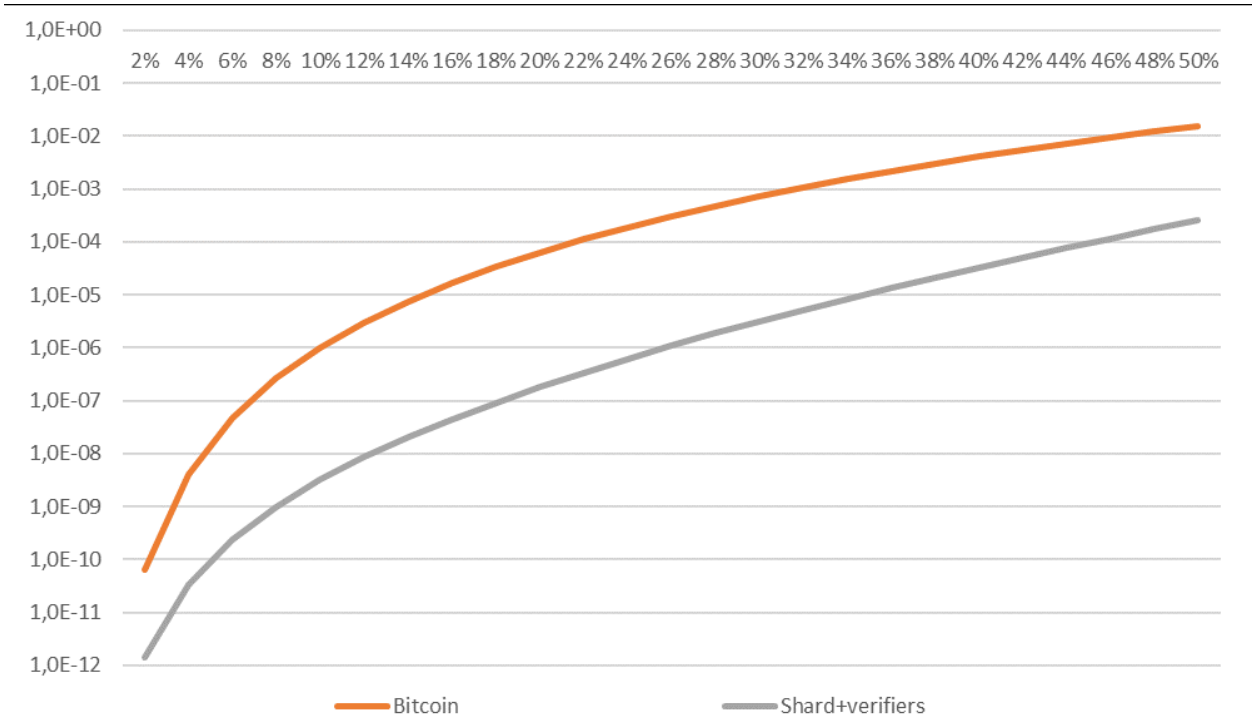
of validators it got receipts from. The Validators that are not sending Block Receipts over some period of time or within a particular delay must be slashed.

106. The MC will verify the broadcast proof message by decrypting the block hash using multiplication of pubkeys of validators which provided receipts and comparing it with the a collator's block hash[25]. Passing the check for 51% of validators will finalize the block if no NACK messages have been received so far. Assuming there are 200 neighbors and that there are less than 50% of malicious validators, getting 50%+1 receipt guarantees that there is 2^{-200} probability for broadcast not to happen.
107. Now that we proved 51% of validators got the block, let's choose a set of verifiers.
108. In order to do this let's choose a random number and a secret that only the verifier knows. Let's then calculate if the validator is a verifier based on those two numbers which would be random and could not be calculated by an outsider.
109. When receiving the block candidate broadcast, the WorkChain validators will calculate Ω from a thread ID, block candidate sequence number and hash of a masterchain block signed by a verifier private key. Calculate H_v = hash (Ω). Let N be a number of validators in WorkChain, R is a predefined parameter in a workchain's config. Then T is a multithreading factor equal to [N/R]. If a remainder of H_v/T division = 0 then the Validator is a Verifier for a thread block.
110. If the Validator is a Verifier it will verify the block immediately and send the result (ACK or NACK) to all Masterchain Validators and to a certain number of validators in all other Workchains together with the hash of the block candidate and the proof of correct Verifier selection.
111. If a Verifier will not submit either Nack or Ack it should be slashed. In order to accommodate that we could imagine a protocol that would reveal the true verifiers after the block has been finalized. Several such schemes could be imagined where the Verifier will have to calculate its participation from some key that must be revealed in another context. Since all ACKS and NACKS are included in the masterchain block it is available publicly. The Broadcast protection receipt can include a validator signature that would reveal the key from which the Verifier has been calculated.
112. Now Masterchain validators will need to wait for a thread block signed by 51% of that thread consensus and broadcast proof from at least one of Broadcast Protectors. If no NACK has been received for a certain period of time (which in reality is the broadcast proof creation time), the block header will be included into the Masterchain. If at least 1 NACK message has been received, the Masterchain will issue a special verification procedure.
113. Since in every round the probability of any particular validator to become a verifier is 1/T, the probabilities for each verifier are discrete.
114. Malicious validators (M) attack could only be successful if the majority of thread validators are dishonest, none of the honest validators are chosen as Verifier and their message delivery fails.

[25] https://crypto.stanford.edu/~dabo/pubs/papers/BLSmultisig.html

Disregarding the network failure rate the probability of this event was compared with failure of Bitcoin “6 block finality”.

115. Let’s compare this with a probability of similar attack on Bitcoin for C =4, R = 15, N = 900



116. We have proved that by having a mechanism of block verification by a random validator set from a global set we can decrease the threshold of the consensus validation in the thread (effectively to zero), reducing the amount of time necessary for block finality while simultaneously greatly improving the thread security. Applying the correct economics to that model we have also proved that it will be completely impractical to corrupt any block data.
117. We can also prove that by reducing the BFT assumptions in the thread we mitigate most attacks on that consensus. In BFT consensus if the thread validators could not come to the consensus regarding the block, i.e. more than ⅓ of its validators are malicious, they could be easily slashed and excluded from the thread validator set by masterchain validators based on the verifiers' attestations of the collate candidate. Moreover we could extend the use of that mechanism to the Masterchain blocks as well and therefore ensure that even the masterchain could not be attacked by a ⅓ of malicious validators. In fact, the only thing BFT is good at is fast coordination of an execution, providing necessary redundancy.
118. By proving that only one node could be enough to Reject a malicious block to stop the attack we demonstrate great improvement in security assumptions of proof-of-stake networks making it a more secure network than a proof-of-work.

119. Question remains, could we make it even more radical and provide a zero probability network security guarantee? What if we add a full node capability of producing Nack messages as in Dynamic Fisherman with one critical difference. In fisherman the proof of a block corruption is supplied post factum, rendering the whole idea practically useless, as we discussed above. Adding the ability of a full node to deliver a Nack message within the block producing delay changes everything. Now we only need to prevent DDOS attacks on both the full node itself or by a full node. In order to do this we can imagine a simple commit reveal scheme with a deposit bond. The full node which wishes to check blocks for validity (lets call it a Watchdog) will create a contract in the Masterchain using some private key and post a bond there. Once a malicious block candidate is detected, such Watchdog will send a Nack message and a bond contract address, signed with the private key of a bond contract holder. Now if proved wrong Masterchain validators would know who to slash and nobody would not know who are the holders of the bond contracts before the Nack is actually received.
120. Such a mechanism would not be incentified and the fisherman-type incentives are useless because there would be no corrupted blocks detected ever, since the attack will have practically 0 chance of succeeding, as we demonstrated in this chapter. Yet since there are many parties which hold a lot of tokens, altruistic behaviour is expected in this case as it will guarantee the security of their tokens to be virtually 100%.

Masterchain Slashing and Recovery Protocol (MSRP)

121. Let’s now consider MasterChain security. In Everscale EK 1.0 there are up to 100 nodes in the Masterchain. BFT consensus protocol with 66% of validators does not have enough security guarantee for a whole network to rely upon. For example if the network value is 10 bln and say 25% of all monetary supply is validating the network security guarantee is roughly 300-400 mln. that can be easily distributed between very few powerful entities. There is a big difference between a network secured by all its tokens, let alone its hashing power, and a network secured by a couple of entities. Fishermen won't help us here (yet again). If the Elector contract is running on the Masterchain the collusion of Masterchain nodes won't be prevented by submitting blames into a contract those same validators run. They will simply ignore it, change the contract and do whatever they want if that is what they are up to. There is a possible argument about social consensus and forks, etc, but we need to compare this network security with at least a Bitcoin which does not need to do any of these assumptions. By partitioning the network we are weakening its decentralization properties, now we need to correct this. As previously shown with the SMFT consensus protocol, we have increased the security of threads significantly.
122. **Therefore a protocol must be created that ensures — if any of the nodes in the network provides a proof of a materchain block invalidity the action can be taken by a simple**

majority of all validator nodes in the network regardless of exactly which Workchain they are currently validating. Let's make several assumptions:

a. at least 51% of validators' stakes in the network are honest
b. all nodes in the network must receive all Masterchain blocks
c. validators representing 51% of stakes can ban any validator from the network regardless of what chain the other 49% are currently validating

123. Once 51% of validator stakes realise the majority of Masterchain validators are Byzantine they need to send a blame with proof to smart contracts which are not controlled by a Masterchain. This kind of assumption dictates a very different Elector contract design. Such an Elector contract should be distributed; its addresses reside in the chains where validators owning them reside. The distributed consensus should be formed between these smart contracts. Without such a consensus no Workchain or Masterchain could function. In our example, if a Masterchain Elector contract can not form a consensus with other Workchain Elector contracts, they will simply elect new Masterchain nodes and continue without the rogue validators.

Distributed Dynamic Validator Set (DDVS)

124. In order to implement MSRP protocol the Election process of the Everscale network must be organised in the following way:
125. The Election mechanism should be replaced by a DDVS, which is a constantly running Election consensus mechanism with no epochs or elections. Elections in general are bad by design. Apart from putting stress on the network by creating unnecessary heavy iterations over a single ledger. It also makes it difficult to implement dynamic slashing mechanisms and despite the fact that it has been implemented already, we believe a distributed smart contracts implementation of the validator set is essential. Additionally, such a dynamic validator set is needed to allow MSRP.
126. This works as follows: anyone wishing to become a Validator will deploy a DePool Root Contract at any Workchain they want except for the Masterchain. The contract will have several methods, such as, “accept stake”, "stake lock", "slash" etc.
127. DePools are a liquid stacking protocol invented in Everscale; it preceded any liquid stacking solutions that later appeared on other networks. Most capital in Everscale is currently staked through DePools. But the first version of the contract had some limitations by design. Let’s talk about the new DePool design and about the new Elector contract design.
128. The DePool Root Contract is a Root for DePool Wallet contract. Any user can ask a DePool Root to deploy them to a DePool Wallet. Once deployed, this user will be able to deposit their funds into this contract. DePool Wallet can accept the user's funds and then users can choose to Stake their funds with any DePool Root Contract deployed by a specific validator. The user coin never leaves the DePool Wallet, as it will only send a message to the DePool Root Contract that the user has chosen with a confirmation to the stake; if DePool Root accepts the

stake it will respond with "Lock" instruction. The Lock period is a discrete constant for each DePool contract. It also includes the set unlock period for each stake to leave the network.

129. Now we have a lot of tokens locked in user DePool Wallets and some DePool Root Contracts which accepted some of the DePool Wallet stake requests and locked their tokens. We need some distributed mechanism to change the Validator Set in the network configs.
130. Let's have a copy of the Distributed Elector & Config Contract (D'Elector & Config) sitting in every chain the network has. In fact, let D'Elector & Config be required to start any Everscale chain (whether Masterchain or Workchain). In order to become a validator or add/withdraw stake, the DePool Root Contract will send a corresponding message to the D'Elector & Config contract requesting the change. D'Elector will receive such a message, verify that DePool Root is an authentic contract (see Distributed Programming for more details on how that's done), execute config parameter change and send a corresponding message with the new config to all other D'Elector & Config contracts on the network.
131. Other contracts will vote for a proposed change with the stakes they register. If the proposed change receives more than 50% support, the nodes will execute the configuration change.

Improved Finality

132. In practice, the collated block that has been broadcast and that is correct can be considered final because the collator and the validator of the thread will have too much to lose. And since the block broadcast to all full nodes in the network, any party that would run a full node and collect a block from collator while himself an honest participant could himself validate that block and consider it final, simply because the fact that their full node has received the block proves that the block indeed has been broadcast.
133. It will be a little harder for a light client to verify that the full node owner did not collude with a collator, it should either wait for a masterblock or collect some number of verifiers ACKs.

REMP — Reliable External Messaging Protocol

134. REMP's objectives are to guarantee delivery of external messages to a smart contract in one WorkChain from any network participant in a particular order and only once. It is hard to overstate the importance of achieving these objectives.
135. In EK 1.0 design a full node transmits messages via overlay groups to the validators which then try to apply these messages to a block without any guarantee of delivery. It has five main flaws: the message is delivered slowly, the delivery is not guaranteed, unnecessary traffic is created, the order of messages in the block is arbitrary and messages could be replayed.
136. The acceptable time of message execution should be measured in milliseconds. If a high frequency of messages are sent the replay is a real issue. Done on the contract level, it complicates code, costs fees and is generally unreliable.
137. REMP is a protocol that guarantees real-time reliable delivery and execution of messages with built-in replay protection.

138. Definition: A **Network Participant (NP)** is any node that has all communication protocols and software needed to access and parse block data, resolve ADNL address of all current and future validators in a shardset, send a p2p message to a Collator and receive an **Accept (AR)** or **Error Receipt (ER)** from it.
139. **Message transmission to thread validators:** Network Participant should broadcast external message to the current and next validator set of a thread depending on destination contract address.
140. In order to do so NP should:
    a. Calculate an ADNL address for all current and next thread validators (from the latest masterblock available to the NP and max parameter).
    b. Send a unicast message to REMP participants, which are calculated as current and future thread validators.
    c. Validators should immediately write the corresponding status of that message into “Message Queue Catchain" (MQC) with a timestamp as a 64-bit time_t integers.
    d. Only the collator will send a signed Confirmation to the NP, once it has successfully collated the block. Collators will write the confirmation to the MCQ as well. If there are more than one collator all of them will send such a message. If the block timestamp collated by a Collator is later than the confirmation, the message must be inside that block. If the message has not been included into the Block by the collator which signed the message, the NP can send such message with validator signature into a special smart contract which will slash all validators of a particular thread.
    e. Validators will slash the Collator if: it did not include all messages it confirmed (soft slashing), it has included the same message twice (hard slashing)
    f. If the block was not collated and another collator has produced it the order of the messages should be as following: all internal messages that were received by the time of the unsuccessful block collation, all external messages received by the time of the block collation, all new internal messages, all new external messages.[26]
141. The order of the messages within a block time slot is not very important; what is important is reliability of message collation in case they were received, the order of messages between internal and external of the current and next block.
142. The next set of validators will naturally create a new MQC which will include Next validators of the current thread. Same thing will happen with Split/Merge. Once the new MQC is created the

[26] In practice this guarantee is harder to implement than it seems for the asynchronous nature of Everscale. Some changes will need to be implemented in the internal messages structures and block formatting. Yet even without this guarantee the REMP will achieve all of its objectives. It is important to understand that this optimization is affecting block optimistic finality time from say 2-3 sec. to 200-300 ms. In the best case scenario. Bottom line it is nice to have rather than must have optimization.

Validators should look at the last masterchain block before the set changed, look at the last shard block that has been written there and compare the untreated external messages in their MQC.

143. They will move untreated messages in the old MQC into the new MQC once they become validators of that thread.
144. If a thread is split, both queues should split as well. If it merges with another thread, queues should merge as well.
145. If the same message arrives more than once it should be discarded by both validator sets on arrival and not inserted into any queue, thus ensuring replay protection. Collator can not collate two external messages which are the same.
146. Collators should try to apply all external messages to a thread block immediately. At least 30% of the block should be allocated to external messages. There could be more included if the block has empty space or less if there are not enough external messages in the queue.
147. There are some exceptions to the REMP messages guarantee, for example the ‘now’ TVM instruction. Since it's dealing with the current real time, the execution of such an instruction could not be guaranteed over some time period, therefore if a REMP message contains such instruction execution the REMP will not be guaranteed (such message sender will have to wait for block finality in the thread and potentially even masterchain.

DDOS protection

148. If an NP is sending too many messages that are not accepted, the shard validators will ban such NP for a particular period of time (cooling), they should not try to execute messages from said NP for a period of a ban, yet they should write a message about their decision to ban NP into MQC together with time duration of the ban. They should notify NP about the ban by sending a special Error message to the first 100 messages received from such NP. The Error messages should include the Ban code and time of its expiration.

Chapter Three
Ever Operating System

> An operating system (OS) is system software that manages computer hardware, software resources, and provides common services for computer programs.
> — from Wikipedia

About Ever OS

149. Ever Operating System is an intermediary between a user and a blockchain — a distributed verifiable computing engine.
150. A modern blockchain like Everscale is not just an immutable ledger. Bitcoin and other earlier blockchains were mostly ledgers, yet even Bitcoin supports a non-Turing complete script that provides some transaction execution instructions.
151. Most blockchains after Ethereum are, in large part, distributed computing engines that execute and verify Turing-complete programs called smart contracts. In simpler words they are a special breed of network processors working in orchestration (called "consensus") to perform common operations and in that way verify the correctness of their execution.
152. In Everscale this paradigm is taken to the extreme. The immutable ledger is quite a small part of Everscale. Of course it is an immutable ledger and a chain of blocks — that is how the data is written and transmitted from one network processor to another — yet there are at least two aspects which make Everscale uniquely more so a computing engine than a simple ledger.
153. Almost everything in Everscale is smart contracts. Every account in Everscale must be associated with a smart contract code (or **initialized**) in order for a user to be able to perform any operation with it. Smart contracts are Everscale Assembly programs executed in the Everscale Virtual machine much like any assembly code is executed by hardware or by a virtual processor in a regular computer.
154. Between a regular computer and a user (which may be a developer who would like to write programs for that computer or a regular user who would like to execute and interact with these programs) there is something called an operating system.
155. That is how GNU defines an operating system:

    > Linux is an operating system: a series of programs that let you interact with your computer and run other programs.[27]

[27] https://www.debian.org/releases/buster/amd64/ch01s02.en.html

156. An operating system consists of various fundamental programs which are needed by your computer so that it can communicate and receive instructions from users; read and write data to hard disks, tapes, and printers; control the use of memory; and run other software.
157. It is quite obvious why computers need an operating system. Before operating systems existed, interaction with computers looked horribly unpleasant to the end user. Something resembling today's interaction between a user and a blockchain.
158. Any way you look at it, blockchain is quite a good candidate to be called a decentralized computer. At least some of the blockchains are. Everscale most definitely is.
159. And just as with any computer, a blockchain needs an intermediate layer (or layers) that manages its resources and provides services to the programs the user runs or interacts with. Of course blockchain, in terms of architecture, cannot perhaps be compared directly 1:1 with a regular PC. But in logical terms, whenever we think about a software stack needed to enable interaction with a user — to call it an operating system is quite compelling.
160. Let's run some arguments. For reasons of practicality we will not talk only about the Everscale blockchain, but most of the arguments could be applied to some other modern blockchains as well.
161. A classical operating system is expected to provide: Memory Management, Processor Managing, Device Managing, File handling, Security Handling and so on. In this chapter we will discuss how all that is implemented on the blockchain for the first time.

File System

162. In Ever Kernel the address of a smart contract is calculated by hashing its code and initial data. The full address, consisting of a 32-bit WorkChain_id, and the 256-bit internal address or account identifier account_id inside the chosen WorkChain. In operating system terms it provides address space management functionality.
163. In the context of the Ever Operating System though, The Merkle tree of Ever Kernel 1.0 provides just part of the necessary functionality to build a fully distributed file system. Therefore we are adding two additional search trees in which nodes would represent contract code hash and contract data and leafs would be contract addresses. We are optimising for fast lookup for contracts with similar data or code hash from within the Node and adding subsequent instructions to TVM to allow this lookup from within smart contracts. Additionally, we add code versioning within these trees thus allowing following the evolution of a smart contract code after setCode operations.
164. This functionality will be particularly useful in the Distributed Programming Paradigm (see below).

File names and directories

165. The Ever OS user should be able not only to call a program by internal processor address[28], but to use human readable names, store data not only in the contract internal memory but have access to some peripheral devices such as hard drives, long term storage and so on that would represent a natural functionality of an operating system kernel.
166. File names and directories have been implemented by a protocol we call DeCert (Decentralized Certificates) in general and in particular DeNS (Decentralized Name Service).
167. The implementation of DeNS is an example of the Distributed Programming Paradigm of Everscale (see a special chapter below for more information) which provides an instant name resolution.

Tonix

168. Following the above a practical simulation of a UNIX filesystem has been implemented[29]. Tonix provides basic file system functionality, as well as an interactive shell with a Unix-style command line interface. The following categories of operations are supported: query file tree and file system status, manage user session, operate on files and file attributes, manage devices and file systems, manage user access, process and format text, browse reference manuals[30].

Storage and other Peripheral Workchains

169. Apart from dynamic memory, which is supported in the form of hashmaps and key-value databases, which Everscale as a whole in fact is, there should be a Permanent Storage for large amounts of raw data. This storage should be accessible by an external user as well as by smart contracts running on EK. It should support common file-system data operations (such as read and write) as well as metadata operations (such as create file and lookup).
170. There are many known problem of distributed storage architectures: as per CAP Theorem, we either store the data on every node of our consensus therefore ensuring availability and partition tolerance (the system continues to operate despite an arbitrary number of messages being dropped (or delayed) or we start to optimise for consistency (every read receives the most recent write or an error) thus trying to reduce the number of nodes holding the stored

[28] “consists of a workchain_id (a signed 32-bit big-endian integer defining a workchain), followed by a (usually) 256-bit internal address or account identifier account_id (which may also be interpreted as an unsigned big-endian integer)” — Telegram Open Network Blockchain, Nikolai Durov

[29] Tonix — UNIX filesystem simulation on Everscale — Boris Ivanovsky, TON Labs

[30] https://github.com/tonlabs/tonix

data, reducing therefore partition tolerance and availability as with smaller amount of nodes there is much larger chance for them being off line or corrupted.

171. Economically the cost to store a file on a hard drive is constantly decreasing. The most cost validators pay today is for the internet traffic. Therefore it is the partitioning tolerance that costs the most. The more validators relocate the data — the more it costs.
172. Ultimately the cheapest way to store a file would be to contract a particular validator and store it on their hard drive once. Of course this will be at the same time the least secure option.
173. The problem is, when we talk about decentralized storage we are not just talking about some consensus over a stored data ensuring censorship resistance and safety. We also need to be sure the data is actually being transmitted to its consumer whenever requested.
174. **A protocol is needed to ensure that the data is stored continuously for an agreed upon period of time, that this data is protected from attacks on its integrity, that it is private, that it is censorship resistance, that it is verifiably retrievable and that the incentives of network participants aligned with their goals of storing and retrieving the data.**
175. Let's imagine a Workchain where shards are not rotating (or rotating very slowly to avoid collusion of validators using a limited number of storage devices), i.e. having the same set of validators more or less. This Workchain would be optimised for storage. Let's call it a "DriveChain" (will discuss other types of storage such as "IceChain" for cold storage later).
176. DriveChain will execute a Everscale Virtual Machine with some additional set of instructions related to storage. Such instructions will be: **Write Init, Write Receipt, Read and Delete.**
177. The Nodes which would like to join DriveChain as Validators will state their capabilities in terms of disk space they would contribute to the network. The D'Elector contract of the DriveChain will "mount" the Validator into a particular shard (in which case the shard would be more appropriate to call "a Drive" or ``DeDrive). If a validator has fallen out of sync, it must signal this to the DriveChain D'Elector contract. If more than 10% of the validators are out of sync, the D'Elector should start adding validators to the DeDrive.
178. In order to make sure that the validators are committing the disk space, files consisting of pseudo random bits would be written into the disk space to fill all the committed space. Once the real file arrives for a particular account to be stored, the file would be created which will replace random files.
179. As usual the DeDrive validators will produce blocks periodically. The blocks would be added to the global DriveChain state exactly like they are in any other Workchain following the multithreaded approach. Therefore the blockchain data associated with DriveChain will be the same across all the DeDrive Validators, only the Storage data, which is located on DeDrive Validators hard drives, will be sharded.
180. The File is written into the validator's hard drives by chunks of some length defined in the DriveChain config. Validators will construct an Extended Merkle Grid[31] of all the chunks of a File

[31] https://www.researchgate.net/publication/344603589_Merkle_Hash_Grids_Instead_of_Merkle_Trees

with SHA-2 hashes of the chunks for the DeDrive Storage and update the root hash in the DeDrive index smart contract (think of it as a directory index).

181. In each DeDrive block the collator will reveal a File chunk corresponding to a sequence number of a chunk calculated from a pseudo random hash (RND) of the DeDrive Root, hash of the last masterchain block and a sequence number of a collated block (Rnd mod X, where X is the number of validators in the DeDrive). The validators will validate on their own data (running a checksum verification on the file beforehand) that such a hash of a chunk corresponds to the hash of their chunk and sign the block. This information, including the chunk itself will be written into a MasterChain block. They will Reject the block if data is not corresponding.
182. Every masterchain block, a random set of Verifiers on all Everscale Workchain will perform a sampling by requesting a chunk corresponding to the data committed to different DriveChain blocks written into the Masterchain block corresponding to a sequence number of a chunk calculated from a signed by that validators private key hash of a masterchain block (RND mod X, where X is the number of validators in the DeDrive)
183. If the data does not exist or does not match, the validator will be asked to provide additional proofs of different data chunks by a global set of Verifiers.
184. The sampling will be initiated by the same set of Verifiers used in a SMFT Consensus Protocol. Several Verifiers from each Workchain are chosen to perform the sampling. If the chunk is not available or is corrupted and the validator is not in the D'Elector list of "out of sync" validators, the Verifiers will produce a Blame and if the total amount of Blames reaching 3 additional verification will be initiated by the selected Validators Committee. The blame must include the wrong block provided by the validator as well as all other blocks received by the slasher from the DeDrive. All Verifiers will request the corrupted or unavailable chunk as well as an additional chunk corresponding to the last hash of the MasterChain block and the verifier pubkey.
185. The 66% of votes of the Validators Committee will decide on the data availability. If the DeDrive validator has failed to produce chunks for less than 10% of requested chunks it will be slashed by 50% of his stake, if more than 10% is detected the 100% of the validator's stake will be slashed.
186. The slashing amount will go to the Verifiers who detected the slashing conditions in case the Validators Committee acknowledges the failure. If the Validators Committee rejects the failure those Verifiers will lose their stake.
187. Remember that all validators within the DeDrive will store all the data associated with Accounts of that DeDrive. The only difference with a regular thread would be that instead of storing everything in a database, calling storage instructions of the TVM will result in a file being written or read from the disk on that Validator machine.
188. The file that is written into DriveChain has exactly one File index smart contract. When a File is stored on a Validator machine's hard drive, the name of the resulting file will exactly match the File index Address.
189. In order to give a file a human-readable name, the DriveChain DeNS contract should be used which will point into a File index smart contract.

190. When a consumer wants to write data it will need to deploy a File index smart contract on the DriveChain. On deploy the smart contract will call the "Write" instruction with the hash of the file the user wants to commit. In return, TVM (Collator) sends a FileConnector. The User obtains the Connector from the smart contract once deployed and sends a file. The Collator receives the file, and invokes the smart contract with Write Receipt instruction.

File index smart contract

191. Below is an example of how to upload a file into a DriveChain.
192. Let's take a hash of the file and add a contract code to calculate the address. Deploy a File index smart contract with the size and a hash of the file and call its constructor (with the TVM command WriteInit). The contract generates a special action, WriteInit, which stores the FileConnector. FileConnector contains the collator's ADNL address, expiration time and owner PubKey (to check connection initiator). Once a File index is deployed, the user can send a file to the ADNL address indicated by the FileConnector within an expiration time provided, signed by a PubKey of the File index.
193. DriveChain TVM instructions could be called from a smart contract by internal messages. The message is sent to the contract in the DriveChain. It will pay for the execution of a storage operation instruction which includes the storage fees for that file.
194. Pubkey is necessary to validate the upload client to get rid of spammers who can use the upload credentials. To initiate an upload, the owner should send a message signed with his private key. Collator checks the signature with the pubkey and starts uploading.
195. To share the data, the owner would need to give corresponding permissions by changing the File index accordingly. In operating systems the file usually has permissions applying to that user. When a user logs into an operating system it checks the permissions this user has on that file. In the case of Ever OS such permissions would be added to the File Index in the form of an address or a PubKey or a DeCert certificate. The Validators won't transmit the file to an unauthorized party and if they do the Relayer would stop such transmission and slash the shard validator. Of course the validator could just give direct access to that file to some third party, but in that case this would be equivalent to stealing a physical storage device from somebody. Apart from the encryption there is no way to defend against such an attack.

How the rest of the DeDrive validators get the file

196. A collator creates a special CreateFile transaction in the file smart contract and attaches the hash of the uploaded file. The contract will count 66% of all validator transactions to validate the file states as available and issue a WriteReceipt transaction. Now everyone knows the file is available. The rest of the validators must download a file in order not to be slashed when verified.

How to Read the File

197. The Read Instruction will return the hash of the offset the user requested. From this offset the calculation can be made using all validators PubKeys to determine which validators should relay the file to the user. The user will run the same calculation using a validator public key to determine where to connect to get the file. It will open simultaneous connections to all Relayers and send them a signed message with a request for the file. The Relayers will connect with the DeDrive Validators, and send them the user-signed message to receive the file. The Validators will check that the Relayers are chosen relayers, that the message is signed by a user which has permission to read the file and transmit the data.
198. In order to read the data a user will post some Bond into a smart contract. The user gets the Bond back if it sends the proof that the file has been transmitted to him minus the Read cost. A Relayer will receive 66% of the read reward (the rest goes to the DeDrive Validator) if the data has been transmitted, which the Relayer can easily verify by taking a hash of a chunk which has been relayed and checking if it belongs to the Merkle tree of a storage since the file root hash is stored in its index. If the Data isn't available the Relayer will send a Blame. If there are 66% of blames for that data received the DeDrive Validators will be slashed for not transmitting. If the user gets the file, it sends the proof that would release its bond.
199. The Relayers reward for a Blame is much less than its reward for successful transmission, but they only get it if the user has sent a proof of the file transmit. Now it seems there is nobody in the system who is incentified not to store, transmit or lie about the data.
200. One of the first consumers of the DriveChain would be other Workchains which would archive an old state. The D'Elector smart contract will once in a while create a request to store some passed state archives and randomly choose validators who will commit its archives to the DriveChain. The validators who do not successfully upload files will be penalised. The storage costs of the Archives will be paid out from the Validator Giver account. The cost will be offset by reading fees for the archives.
201. The consumer could store any data in the encrypted format. No validator would have the ability to decrypt the data and read it. Relayers will get chunks of data to transmit to the consumer making it de-facto a torrent-like network. There could be a garlic-like protocol implemented on top of Relayers easily as their addresses are ADNL addresses.
202. In order to give permission to modify or read the file, such permission should be written into the DeFile index smart contract.

IceChain

203. There could be a slower Workchain that would store long term archives, let’s call it an IceChain. Such an IceChain would operate exactly like the DriveChain with some modifications. Since the

DriveChain is a Virtual Drive blockchain and the data should be readily available in the current state of the art, the use of Zero Knowledge Proofs (ZKP) may not be practical. Yet in the IceChain where validators can store the files in a long-term Glacier type of Storage (such as Tapes etc.) the access time does not play such a role as the user is expecting to get the file with potentially significant delays. This opens a possibility to use non-interactive ZKPs with no trusted setup for data sampling.

204. The sampling would include constructing a ZKP for a different random number of chunks for each verification whose delay could be set to minutes. That would render the possibility of data corruption in the IceChain to virtually zero. The proofs would be verified by Everscale Verifiers periodically and would contain everything required to blame if the data is not available or corrupted. When the ZKP reached the state of technology that would enable them to produce the proofs at the high rate, they could replace the verification mechanism on the DriveChain as well.

Chapter Four
Web Free

205. Decentralization claim can only be made when the whole system is decentralized and, therefore, should be judged by its “weakest”, read “most centralized” link. For instance, there is simply no point in creating a maximum decentralization on a consensus layer and then passing all transactions through one server.
206. Ever OS is built around the concept of the End-to-End Decentralization Framework (E2ED for short) which also goes by the name “Web Free” as opposed to “Web 3”.
207. The reason we disagree with Web 3 is more or less the same as we disagree with the multi-layer approach. What is correct for the critique of a multi-layers blockchains is even more so for a specific Layer such as Web 3. To add the current Web into a blockchain is akin to adding a rotten apple to the barrel. As we already discussed above the current web should be displaced by an entirely new technology to be brought back to the user. This technology we call “Web Free”.
208. Notable Web Free components of Ever OS are: smart contracts and tools for their development, testing and security; Distributed Programming Paradigm, DeBots, Pipes, code handling, compatibility, bootstrapping and upgradability. They will be discussed in this chapter.

Smart Contract Languages

209. Ever Kernel v1.0 uses the Everscale Virtual Machine for smart contract execution. The reason to use such a virtual machine mostly relates to the compactness of its code and supposedly easier security model. For Ever OS design considerations it was very clear though that in order for such a platform to be adopted by many developers the high level and familiar programming languages should be adopted to write programs for TVM.
210. The special nature of blockchain immutable programs that may hold very large monetary value dictates special attention to its security. There are largely two ways to look at a solution to that problem. One would imply creating new Domain Specific Languages (DSL) to limit the ability of a programmer to make mistakes. We always believed this way leads to the oblivion of such a platform. Therefore Ever OS middleware includes compilers for Solidity and C/C++ languages (using the LLVM framework[32]). In a special note let’s mention the Ever LLVM compiler team work on advancing the comprehensive stackification logic within LLVM community[33].

[32] https://llvm.org/

[33] Towards Better Code Generator Design and Unification for a Stack Machine. *Leonid Kholodov, Dmitry Borisenkov*

211. The security model for writing such programs using general purpose programming languages is described in the next section, yet there is one tiny note of hypocrisy and one larger problem of efficiency in this logical construction.
212. The thing is Solidity and even C/C++ we are using to write for TVM are not exactly general purpose programming languages. Even a proficient C++ programmer can not simply write a Ever OS program in C++ without first understanding quite a lot of limitations that TVM (or any other such Virtual Machine for that matter) execution imposes on a language.
213. For example, the inability to write code like this:

```
int *arr = new int[10];
```

214. Or the notion of memory which is completely different etc, could probably illustrate the point. Of course one would claim that a distributed processor architecture probably requires a special model of program execution. After all, the use cases for such programs are very different from use cases of a local computer or a network server. Yet is it really so? Can we have a fast compact code execution which would be really easy to develop?
215. Michael Franz has developed several interesting concepts that are worth researching in terms of its potential applicability to Ever OS. There is a prototype developed in Ever Labs for translating Solidity into Slim Binary. On average, a code of several Kilobytes is translated to just a few hundred bytes even without index compression and similar optimizations. Such a code could be executed in a separate isolated environment as part of the TVM instruction set or as a non-TVM code.

Programs security

216. One of the goals of Everscale is to make accessible and easy to use tools to create its programs. Unfortunately for blockchain programming this is not enough. Distributed processing demands certain specific properties from a program such as compactness, efficiency and in particular, infallibility. The latter is important since almost every program in blockchain carries some monetary value, thus it lives in a very different paradigm than, for instance, a regular program executing on a user machine locally. Sometimes the value carried by such a program is such that it could be compared with a program operating a space mission. There has been a long established method to make such programs secure. This method is called “formal verification”.

217. Though every program is definitely designed and in principle can work on a finite set of input data and internal state, the variety of resulting space is great enough to forget about testing (checking) the correctness of a program behaviour on each possible value. Here the math comes. The mathematical logic and correspondent type systems allow to “quantify” (apply a quantor before) the term, which gives the possibility to “check” the program behaviour immediately on every possible input and internal state. We usually write “forall” quantor as ∀

and mean that if the statement ∀ x, P (x) holds, property (or predicate) P actually holds for **every** x from some eligible (probably infinite) subset. The possibility to reason and argue about "forall" quantified properties on a program lies deep in theory of programming languages and its interpretation on admissible machines (real or virtual),

218. If everything is so good, why not formally verify every program. Besides a lot of technical still unsolved tasks, there are two bold arguments: it is always very difficult to perform and to realize that it is actually done. Sometimes it is also completely impossible.
219. On the first point, Poincare and Hadamard[34] established that all more or less formally formulated tasks can be divided into direct and indirect (inverse) simply put - correct and incorrect. Direct task is such where there exists an algorithm to solve it step by step starting from initial data, having the correct answer or proven failure to solve (e.g. number addition, or quadratic equation) at the end. The indirect task is usually an inverse task of finding initial data which corresponds to observed behaviour. Often, humans find a way to transform the inverse tasks to direct (quadratic equation still works as example), but more oftenly they fail. Formal verification is the task to find the properties of the given program and prove them in a formal way. No direct way to solve this has been found.
220. But no less critical as we said earlier is that sometimes the whole task is completely impossible. As programs are not scientific objects, their properties have to be realizable by developers and users. Mathematical proof is not of common sense, and even an excellent programming engineer might not realize all the properties the written program satisfies. Moreover the set of program properties is almost always infinite and there is no known general way of reducing that infinite set to a "finite minimal normal form" with an exception of trivial programs like a list sort. Another example is arithmetics. Widely known that number theory is not fully developed and may never be. Kurt Gödel broke people's hopes on fully closed mathematics by stating his theorem of incompleteness[35]. So if a program behaviour is based on some number properties which are still unknown (unproven) or very difficult (Fermat theorem), to prove program behaviour, correspondent theorems have to be involved. However it seems very simple to write a program which depends on the Fermat theorem's truth. Easy to realize that understanding the properties of such a program will be close to incomprehensible for an ordinary observer.
221. The latest point is commonly referred to as a main incapability of formal methods. It lies in the so-called halting problem[36] originally formulated for abstract Turing machines. Informally speaking it means that there is no universal program which can reason about any other input program if the latter uses wide enough grammar. Simpler, one cannot write a program which can prove that any given other program halts or runs infinitely.

---

[34] Jacques Hadamard. Essai sur la Psychologie de l'invention dans le Domaine Mathématique. Paris, 1959.

[35] https://en.wikipedia.org/wiki/G%C3%B6del%27s_incompleteness_theorems

[36] https://en.wikipedia.org/wiki/Halting_problem

222. As a result we see deep incompleteness of any formal method we can use to reason about programs. However the internal contradiction is the following: we have the powerful mechanism to speak about program properties, much more powerful than any testing suite can even imagine but it is limited by the very nature of mathematics. There are many engineering approaches to mix the methods, verifying core parts and testing a less sensitive environment, modelling core program behaviour or underlying protocols in simplified sandboxes and many others.
223. However we think that the root of the problem is that humans search for a more expressive way to write programs where the more expressive the meaning is - the less it is understandable by other programs.
224. Having the given problems the keys to look for a solution are quite simple:

    - every programming engineer knows or believes why her program halts or works properly and she can answer the direct question why it is so - otherwise there is a good motive to rewrite or refactor;
    - nobody really wants to write a program with uncertain properties for production use. Instead of analyzing a program's correctness by external tools, we might give a programmer tools where she can explain why her program should work like this. Moreover in most of the cases it is quite self explanatory.

225. What we propose to have in Everscale for the good of formal methods and making contracts secure by construction:
    - We are not going to limit engineers to express their ideas in the most suitable programming language;
    - “Good lenses”: the tool to look into the state on the stage of programming, not running or testing. This requires adequate blockchain model and REPL driven development where every step is depicted in the nearby and gives the correct feedback;
    - “Good configurator”: the tool to express the willingness of both engineer and designer. This tool is an ultimate part - the better we can be self-explanatory, the better all the next parts will work.
    - “Good programming language”. We actually have it (choose any). We need to add some “steroids” for further steps. Ideally to have a language where we can express a trinity: a specification, a program itself and a proof (which actually Coq[37] does but in a too purist way).
        - This language can be embedded to the prover bidirectionally. If one likes to verify the already written program she can translate it, embed it in the prover and verify in a friendly environment. Conversely, if one likes to write a more strict program, having the immediate possibility to prove its behaviour - do it inside

[37] https://coq.inria.fr/

the prover having minimum differences to “normal” programming process and after satisfaction with proofs - translate it to compilable language, build and deploy.
    - The “steroids” mentioned are the possibility **to do it instantly**, not dividing by phases. The ideal configuration is “write a bit of spec -> write a bit of code -> write a bit of proof -> feedback -> loop”. Humanity can do the last three steps more or less acceptable, but without specification one still is unaware of what she really codes.
- “Good prover”. There are a lot of them, no actual need to have a new one. The problem is to embed the specification and program into.
- “Good automation”. We actually need a solver which helps us to think on the level of abstraction we are used to, solving all easier tasks in more or less invisible mode (in background). That is purely an engineering task: the more code of automation - the more actual automation. Good practice here is to cover most common cases: find the most common pattern of propositions, automate reasoning about it, see what remains. Good heuristics and machine learning will not harm. This part is not so time sensitive.

226. Distributed programming (see below) greatly increases capability to do formal verification of programs simply due to the fact they become smaller and less complicated. Also it significantly increases reuse of formally verified code within the ecosystem.

227. Usually in a blockchain the address is associated with some cryptographic key which means that in order to create a new address one needs to generate a new key pair and calculate the address from its public key. Everscale Address is calculated from initial data and a contract code. Every address in the active (called “initialised”) state has a deployed smart contract code. This code can be the original code from which the address was originally calculated or a different code in which case it means the “setcode” instruction has been used to update the code. If the initial contract code does not have a setcode function in it, the Everscale address will unambiguously correspond to the code. Non-active (called “uninitialised”) addresses can only receive tokens if sent as a message with a special (unbounce) flag. They can not perform any operations.
228. This unique feature of Everscale has several implications. For example a user key can have an infinite number of addresses which depend on the initial data and code of a contract deployed on it. The ability to calculate addresses from the data and the code of the smart contract and then updating these without changing the address, makes the whole Everscale blockchain — an advanced key-value store. The difference with a simple key-value store for example, would be that the address (which is a value in this case) depends on two keys (contract initial data and its code). Each of the keys could be viewed as a separate hash array in terms of the database. That allows the creation of subsets of arrays which would have one unique key.
229. A code associated with a Ever OS address can be verified by another address and therefore be trusted. History of address code mutations including initial data and code that was produced by a known source (cryptographic key or another address) is provided as well (as described in “File system” section).
230. This web of trust allows a completely new way to program a blockchain application. Now we can fully trust the behaviour of a particular address and therefore assume the immutability of its input and outputs. We can distribute almost any business logic across global ledger entries without a need for any nested data structures. The ERC-20 type of contract becomes obsolete and generally a bad practice. To program a hashmap inside a contract would mean to program a database within a key-value database cell. Sometimes those hashmaps are practical, but only to store some temporary, small structures. Think of it as an application dynamic memory. What one would put in a program memory you may store in a hashmap within a contract, if one would normally use a database to store an object — use the whole Everscale blockchain as a key-value database. If one wants to store large files, it is better to use a DriveChain which acts like a distributed Hard Disk.
231. Good question to ask when writing a program on Everscale: would this data structure be better placed in a local memory or in a database? If the answer is the former — use a hashmap, later — create an address.

232. Let's think of a Everscale as a key-value store, where what has been used in address calculation is a key and the address itself is a value. Then if we can find an address by calculating over some keys, we can then retrieve and execute what is within that address.
233. Great example of this is the Everscale TIP-2 Certificate. Usually to create a record of something on a blockchain one would make a ledger smart contract that would consist of name and an address. Of course as with any other such contract the heavier the use of that service the more complicated such a smart contract becomes, in the end one would need to start thinking about sharding it somehow. And that is not trivial at all, or may be entirely impossible.
234. In Everscale there is an elegant solution using a distributed programming paradigm:
235. For example, that is how a decentralized and distributed certificate system is implemented in Everscale. As described in more details below, such a system is used in providing many services which require a certified provable key-value store. For example, a Decentralized Name Service (DeNS) is used as a basic filename and directory structure in the Ever OS filesystem. Other uses include a Proof of Ownership / Prove of Purchase certificate and many others.
236. Current solutions (for example a Everscale DNS[38]) are either large smart contracts which maintain a full list of records, or a tree-like solution which shards the list based on some parameters. Neither of these solutions are satisfactory due to a lack of scalability, high costs of maintenance, long search time, single point of failure and so on.
237. Let’s take a look at TIP-2 implementation. Root is a smart contract that contains a Code of Certificate smart contract without data. Root has methods for Certificate Issuance, Certificate Code Retrieval, Root PubKey retrieval and Version history. Each Certificate can become a Root, therefore a Root smart contract and its Certificate smart contract are the same. The Code contains an address of its Root, therefore making it immutable.
238. When a User wishes to register a certificate, it calls a Certificate Issuance method in Root, sending a Certificate Data (for example an alphanumeric string of a certificate body).
239. Root takes its Public Key and a Code of Certificate smart contract, inserts a Certificate Data sent by a User, calculates the address of the Certificate and checks if the address already has a Certificate or any other Code deployed by sending a bounced true message calling the “getData” method.
240. If a contract exists it means that a Certificate with the same Certificate Data already exists. The contract can then return registration information to the Root which will return it to a User. If a contract does not exist the message will bounce to the Root smart contract which would mean the Certificate can be registered.
241. If a Certificate does not exist, the Root will Issue the Certificate by deploying the Certificate Contract with its Data. On deploy the Certificate will check that it has been deployed from the root address by comparing the address of a Root inside with the deployer address. If there is no match the deploy will fail.

[38] https://test.ton.org/DNS-HOWTO.txt

242. Of course additional business logic steps could be included between the last two steps, such as monetization or other mechanics as shown below in one of the examples.

Resolving

243. To resolve the Name, any User can now call the Get method “Resolve” of a Root locally to obtain an Address. Root will use the Certificate Code, Root PubKey, insert a name the User wishes to resolve into the Certificate Code and calculate the address.
244. A user application can cache the Certificate Code smart contract and Root PubKey once, after which resolving any name is achieved locally with a simple address calculation, with no need for network connection at all, therefore making it the fastest certification system in the world.
245. Knowing a Certificate Code hash enables retrieving all smart contracts having the same hash by simply querying the blockchain state. Decoding contract data will produce a full list of names under a specific Root. It would be quite easy to produce a table with all the certificate records.
246. The Certificate itself contains variable types of addresses of target smart contracts to which the Certificate owner wishes the name to point. A user should choose which type of address they wish to use.

Reverse resolution via Index smart contract

247. A lot of times even the advanced key-value store won’t be enough. For example a user deploys some multisig wallet with several custodians public keys. Now how could that user keep track of all such multisig contracts she deployed? And even more interestingly, how the custodians she has added could keep track of all multisigs they are custodians in.
248. Let’s deploy an index for each of the multisig owners. Initial data will have an owner account address for which the index is created (which will be used as a pointer to a contract we are looking for) and the code will be an index certificate code plus the custodian public key.
249. Now we only need to know the certificate code (which is public information) to add a public key to collect pointers to all target smart contracts.

DeBots

250. DeBot — is a smart contract facilitating conversation-like flow communication with a target smart contract. Instead of a usual smart contract, DeBots are executed locally. They work with

target smart contracts providing a user interface to their functions. DeBot can work with one or many smart contracts, DeBots can also invoke each other, transparently to the user. DeBot prepares a message that it will send to a target smart contract (or smart contracts), but before it does it will emulate all the chain of transactions on a local user machine verifying and letting the user verify its correctness. DeBots are extremely powerful user technology, simplifying for the first time complicated interaction with a smart contract or systems of smart contracts.

251. DeBots are completely decentralized by nature. They do not require any servers in between full nodes and a user device. There are no centralized or decentralized layers which clog user experience, like Web 3. Since DeBots run locally for all its phases, except for final message sending, they work much faster than any web or decentralized application. Using Pipes, DeBots can be presented to the user as a single action, interactive action flow, a form, a web page and so on (see below).
252. Users are interacting with DeBots via DeBot Browsers. Browsers contain the DeBot Engine (or DEngine) which takes care of running DeBots and providing D'Interfaces support.
253. D'Interfaces allows DeBots to receive input from users; query info about other smart contracts; query transactions and messages; receive data from external subsystems (like file system) and external devices (like NFC, camera and so on); call external function libraries that allow to do operations that are not supported by VM. For example, work with json, convert numbers to string and vice versa, encrypt/decrypt/sign data.
254. Every DeBot browser should strive to support D'Interface standards which are proposed, discussed and finalized in DeBot Interface Consortium (DeBot IS Consortium)[39].

Pipes

255. In Unix Pipe (or Pipeline) are some processes chained together, so that the output text of such a process (stdout) is passed directly as input (stdin) to the next one. The concept has two reincarnations in Ever OS.
256. Smart contracts that pass messages between each other, a design sometimes called "composability", are actually a pipeline. Everscale composability is built in because of its asynchronous nature, mandatory delivery of messages in a particular order and the nature of all addresses having a smart contract code associated with them.
257. But there are also pipelines in DeBots, so users can create DeBots as a single scenario that could be activated and finished possibly without any user interaction. We call this type of DeBot scenarios — Pipes.
258. Using Pipes, third-party applications can call DeBots as services in a non-interactive mode (or in semi-interactive mode) and receive responses as a result.

[39] https://github.com/tonlabs/DeBot-IS-consortium

259. This allows users to compact the DeBot interaction into a single action (like tapping on a button or double clicking). Now applications can draw a custom UI, collect inputs from the user and then run DeBot's certain scenario: read blockchain data, send on-chain transactions and so on - and receive answers.
260. DeBot Browser can run as a standalone instance without UI components. Browser should isolate DeBot communication with the user and automatically insert necessary inputs for DeBots using the DeBot Pipe. DeBot Pipe defines which function to call to start DeBot (by default, it is the start function without arguments) and what values should be passed to DeBot on each interface call or approve request.

Code Handling

261. Git, a creation of Linus Torvalds, the father of Linux, revolutionized the source control systems space in 2005. Most of the software development has been moved to Git since. Usually organisations host their own Git installation or rely on cloud providers such as GitLab for their repository management. This model did great things to the whole software industry in general and open source in particular over the last 15 years.
262. For centralized projects, even if they are open source, the centralized Git repository may make sense. Though we would argue that virtually all free software projects in fact require a decentralized organization, decentralised governance model, through the DAO with some form of Meritocratic Token Distribution. We argue that without these components, managed by a foundation or just a group of independent contributors these projects can not be regarded as true Free Software. Today pure Free Software projects must be decentralized, or it just does not make any sense.
263. Moreover if we are talking about decentralized projects and their code it is simply absurd for them to use a centralized Git repository. It presents risks that nullify all their decentralized governance efforts. Yet almost all blockchain projects are holding their code in a centralized repository today, and therefore can not claim decentralization as one of their properties. The time will come when software repository content will be subject to the same censorship as centralized internet today. Like social networks ban political figures or just simple citizens expressing their contrarian views with fake-morality arguments, the time will come when repository code will be censored for the same reasons.
264. In this section we particularly are talking about how these and other DAOs should manage their code to avoid centralization and censorship.
265. Git development today requires a centralized management of repository rights. With decentralized tools repository keys could be dynamically managed by a set of smart contracts.
266. Governance tokens could be distributed among project stakeholders according to their ever-changing rights. Such tokens on top of repository management abilities will give its holders a stake in the ecosystem such a project aims to create.

267. Tokens can be minted based on inherited repository performance metrics, such as forks, followers, external and internal commits, reviews, releases and so on. Important decisions regarding the project's future would be taken using decentralized governance tools developed in Everscale.
268. Once an ecosystem is created and growing, such tokens can then naturally transform into the project's cryptocurrency supporting any monetization model the project chooses to adopt. Helping rewarding long-term contributions to the repository as well as inclusion of new members and investors.
269. To support such decentralization with immutability of the Git repository, an append-only Git database can be moved to a high performance, low latency Everscale blockchain entirely. In fact, without such a move no project could claim decentralization because of a centralized way its code is managed. Simply speaking, putting Git on Everscale should be viewed as a must-have Everscale objective.
270. Fortunately there is no problem using Ever OS to build a decentralized Git. The tree structures work very well with Ever OS and blobs of code can be easily stored on DriveChain.

Upgradability

271. There is no problem putting a code of a protocol upgrade to the network and announcing the availability of it within some block. The problem starts when such a code is not backward compatible. The network partitioning will evidently happen as a result of such code spontaneous upgrade of network nodes.
272. To tackle that, the Ever OS upgrade procedure should be as follows. The code should be pushed into the production branch in the DeGit repository which will upload it to the DriveChain and upgrade a smart contract responsible for network code upgrade. Once validators detect such an upgrade is available, they should start a fresh node with such an upgrade, sync with the necessary WorkChain and a MasterChain and signal their availability for validation to the D'Elector contract of choice. The D'Elector will launch a pre-validation procedure that will verify fake blocks with new versions until the necessary number of such nodes are available on the network. Whenever that happens, the D'Elector will issue a key block in which the new validator set becomes the active one. The next block after the keyblock will have a new format, a new version and no blocks produced by the old software will be accepted.

Conclusion
The Evolution

In this paper we presented a system with practically unlimited scalability, practically sub-second finality and practical security guarantees of more than that of Bitcoin. It has the capacity to run any Internet application including messengers, social networks and emails we have today in a censorship resistant, decentralized way. It can grow a validator set to hundreds of thousands of nodes, while preserving all its performance and user experience characteristics. We also describe an economy and governance model to accommodate further development of the network without sacrificing decentralisation.

On 30 of August 2021 a little over a year from the launch of Everscale the internal network running the latest version of Ever OS broke the world record of transactions per second reaching more than 53,000 tps of real smart contract executions. The Kernel configuration that was used consisted of 150 nodes running 10 Workchains configured for 32 threads each, using 1 Gb connectivity. Next day more than 350 independent validators residing in different datacenters joined that same network and set a record of appr. 45,000 tps. No network before has ever reached such numbers, but what it has really proved is that the Ever OS design can scale to practically unlimited capacity with the increase in number of validators via dynamic Kernel reconfiguration.
The above numbers show that Everscale network design can accommodate decentralization of all the current and future internet applications.
We have described that to achieve decentralization the blockchain technology must be inseparable from its economy and governance. The governance and economics model presented here achieves malleability by a set of adjustable rules executed by the community of users, hodlers and developer-entrepreneurs. BFT Governance set of protocols ensures flexible self governance of a protocol and eco system development. Economic system has two token models: EVER — a token that captures value of all community sponsored projects, therefore optimized for a value capturing, while NEVER is its binary companion to be used as a payment token while constantly losing value, therefore optimized for medium of exchange property.

# # #

Once when my daughter was young she got a gift. It was a beautiful tropical butterfly. It had very large wings. When opened they resembled a flying owl head, when closed an incredibly precise picture of a sneak head. The camouflage was of course the result of millions of years of evolution. All butterflies whose camouflage was wanting have died. We as humans possess a brain. It tweaks natural selection into our favour by intelligent decisions. A great tool, if used correctly. Unfortunately no human brain always makes only correct decisions. In fact, our brain

was enhanced through evolution by natural selection of the buoyant while people who made wrong decisions died, like butterflies. In fact any human will make lots of wrong decisions during their lifetime, sometimes deadly ones. Some wrong decisions will be deadly to a whole group of people if made by their leadership. The democratic process of power delegation tries to mitigate a decision by a group by choosing a leadership that would make less mistakes. Many times this process fails simply because the decision of many does not prevent it from being wrong. Look no further than the German elections of 31 July 1932. The only way to enhance the survival of the human race and avoid more deadly mistakes is to stop delegating vast amounts of power to small groups of leaders for arbitrary decision making.

Decentralized Operating System is a technology that allows people to reach consensus in discrete, spontaneous governance groups about arbitrary subjects formulated as software code without a need to trust each other. It enables us to avoid a need to delegate power and to accept the fact that many decisions simply can not be predicted as either right or wrong at the time they are taken because of our inability to look into the future. Decentralization is a framework to make governance mistakes without global consequences.

Acknowledgements

There are dozens of people, hundreds of authors and thousands of Everscale community members who have directly or indirectly helped write this paper. Here I would like to mention by name just a handful of people who had a direct contribution to the ideas, implementation and formalization of different concepts described in it.

Because of the sheer volume of the topics to cover, this paper should be viewed more like a framework of ideas and design blueprints to be further discussed in separate documents, some of which are already available (links to them are provided throughout the paper) and some are still under development.

I feel privileged to work with most talented people in their respective areas:

Pavel Prigolovko, Andrey Lyashin, Dmitry Shtukenberg, Leonid Kholodov, Kirill Zavarovsky, Nikita Monahov, Anton Serkov, Andrew Zhogin, Slava Belenko, Boris Ivanovsky, Sergey Yaroslavtsev, Victor Bargachev, Igor Kovalenko, Alexey Shistko, Michael Skvortsov, Michael Vlasov, Ekaterina Pantaz, Vasily Selivanov, Luca Goroshevsky, Ivan Suvorov, Alexander Novikov

Special thanks goes to all those who contributed to editing: Eugene Morozov, Joanne Eberhardt and Benjamin Bateman