everscale

Everscale Lite Paper

Version 1.01

December 2021

This is the Everscale lite paper in plain English for the layman by Mitja and Luca Goroshevsky. For a precise and thoughtful description of Everscale's technical design, please read the Everscale WhitePaper

Introduction

A decentralized global blockchain network, Everscale was launched on May 7, 2020. On November 10, 2021 by the decision of its community it was given its current name. The backbone of the Everscale blockchain is the Ever Operating System.

Ever OS is distributed and decentralized, that is to say, operating over the internet, through many computer machines. While we will discuss its architecture in further detail later, let us first establish some basics.

A blockchain is a network which needs to execute blockchain programs, called smart contracts, and programs are always executed on a processor. Typically they are found in your PC or on a server, and blockchain programs operate by the same principle, in a virtual way. However, there is one fundamental difference: every program that runs on this virtual processor is also run by many other processors across the network. The result of the execution of the program is compared, and only if all of the network processors agree upon the result of the execution of the program is this execution written into the shared memory across all participating devices, in a 'block,' forming the blockchain.

Once this is written, the execution becomes immutable, a function guaranteed by the immutability of the whole blockchain. The most important property of this immutability is decentralization. If the network is not decentralized, it means some party controls most of the servers executing the program. If they don't like something, or have any reason at all to change it, they can alter the record of the program execution or the execution itself. This means it cannot be immutable. In the blockchain world, combating this is called censorship resistance; meaning no single party, however powerful, can alter anything about the execution of a blockchain program after it has been completed.

These are important concepts that form the foundation upon which the blockchain is built, and without which there can be no blockchain. We will talk about this further. We will also talk about the architecture of this processor; the consensus protocols of the network, and how we guarantee that it is really decentralized; about how the programs are organized; the underlying economy behind it all, together with its importance; and about the community and its system of governance, created for the sustainability of the decentralized global computer. Oh yes, because in talking about all this, we are talking about building a decentralized global computer.

Chapter I.
The Proof-of-Stake and a Validator

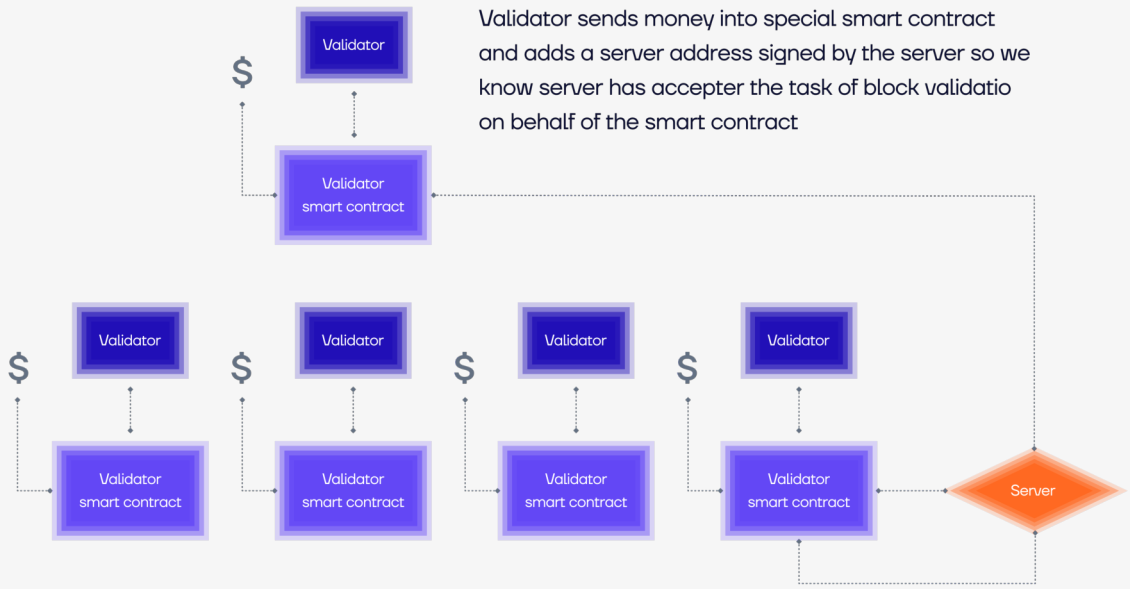
The Everscale blockchain is a proof of stake blockchain. Proof of stake is a system where people called Validators ensure the correctness of the blockchain by validating blocks. They put down a stake which they would be afraid of losing if they fail to do this, and earn a fee if they succeed. The stake, therefore, is a bond which guarantees their behavior. Conceptually, it's quite simple, but when it comes to the implementation of such a system, it's deceptively so. Let us begin at the beginning.

To become a validator of Everscale, you need a computer and you need to have a stake. The Validator puts a stake as collateral and the computer is used to validate blocks. Because Everscale has a high throughput demand it should be a pretty powerful server, but even powerful servers don't cost too much these days, with prices of around €300 a month being common. The server then runs the blockchain programs, and compares the results of these executed programs with other validators. They then need to agree on the results of these executed programs with each other, or in other words, about a common state of the blockchain. They need to have a consensus. We'll talk about how the consensus protocol in Everscale works later.

So going back to becoming an Everscale validator, a user needs to deploy a special smart contract and use it to stake some tokens. Then the server we mentioned previously needs to sign a special message and send it to the smart contract saying that this computer is ready to be a validator for that stake. The stake required could be pretty high (right now it runs at around 350,000 Evers and increasing). Obviously we don't want to exclude people from validation, people who don't have these amounts, and want someone with as little as 10 Evers to participate in the network security. So how can they do that? Well, they can deploy a smart contract, put down their stake of 10 Evers, and then ask someone who has a server to Validate their stake.

Let's say there is a market where they can ask some professional validator who runs servers to validate on their behalf, or more precisely, on behalf of their stake. While there are risks associated with this, the protocols described in this paper are precisely there to eliminate them. Nevertheless, always choose your validators wisely.

The validators servers (called nodes) can validate for many such stakes simultaneously. Once the stake has been accepted they are included into a 'validator set' by all of the existing validators. If the Validator server is new to the network it will be asked to validate some blocks in advance so other validators can be sure that the server is functioning properly. This is called "qualification mode." Once they pass that process the new validator can accept real blocks for validation on behalf of any stakes.



One server can become a validator of many stakes but not too many. The minimum and maximum stakes are datermined try the Ever Os configuration file

Chapter II. Sharding and Multithreading

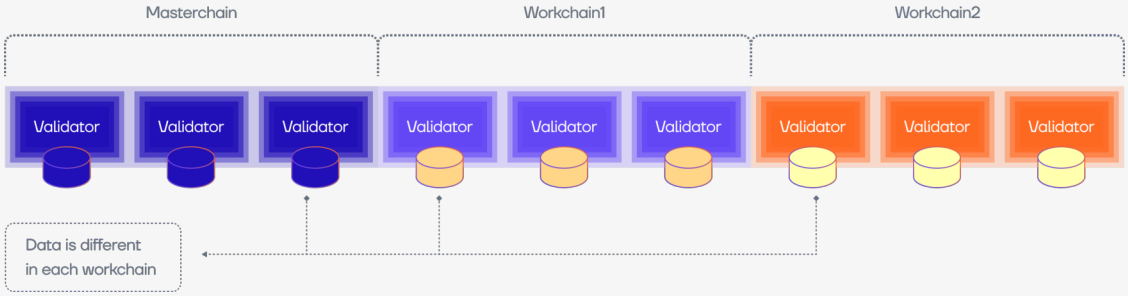
Everscale is the only blockchain which operates through both sharding and multithreading. Let's see how it works.

So if many validators have just joined the network, they need to validate something. So each validator is assigned a chain to validate on. These can be MasterChain or WorkChains. All these chains are separate blockchains, containing many other separate blockchains, as we will see. The difference between MasterChains and WorkChains is that all block proofs from all chains are submitted into the MasterChain. All blocks in a given MasterChain are collections of proofs that the WorkChains connected to it are working correctly. This part of the design is similar to Ethereum 2.0 or Polkadot for instance, but here the similarities end.

Validators are assigned to one of the chains, and they can see all its associated data; they download, store, and change the state of all the programs and some other parameters of this particular chain. This is sharding. Sharding is a separation of data, and the term comes from the database world. And that means that these validator servers do not store the state data of another chain, it's therefor sharded.

Step 1: Validators joining the network

Step 2: Each validator assigned a chain



Yet just to shard the data is not enough to ensure scalability, because sharding of data does not provide the network with the ability of parallel execution of smart contracts that need to be executed on a particular chain at a high enough speed.

There is also a need for a parallel program execution on top of sharded data to ensure true scalability. There are two things that constrain scalability. The first is when there is a need to send a lot of messages between servers: at a certain point the internet connection simply runs out. Once that issue is solved with sharding it's the processing power that starts running out. The solution to this conundrum is something we call multithreading or, basically, parallel execution.

All our modern computers run multi core processors simply because single core cannot execute all the programs in parallel because they just hit certain limitations, namely, the laws of physics. Everscale does the same thing for its virtual processor. That is done with each validator of the same WorkChain being assigned a thread. So there are groups of validators that execute different sets of smart contracts, which are separated by accounts. That's precisely why we have infinite scalability: because we can now add these validators linearly, as more programs we need to execute.

Step 3. Validators inside each Workchain are split into groups called “Threads“. Validators in a thread are rotated every 4 min.

We can add more WorkChains and more threads. And all of that happens dynamically. When we don't need to execute too many programs, there will be fewer threads, and just on the one WorkChain, for example. But once the number of smart contracts which are executed grows and there is a need for more and more processing resources and disc space, more chains and threads will be added.

And, that's how it scales. And that's why it ever scales. And, as long as you don’t run out of all servers in the world and all the internet capacity in the world, Everscales.

Chapter III. Soft Majority Fault Tolerance

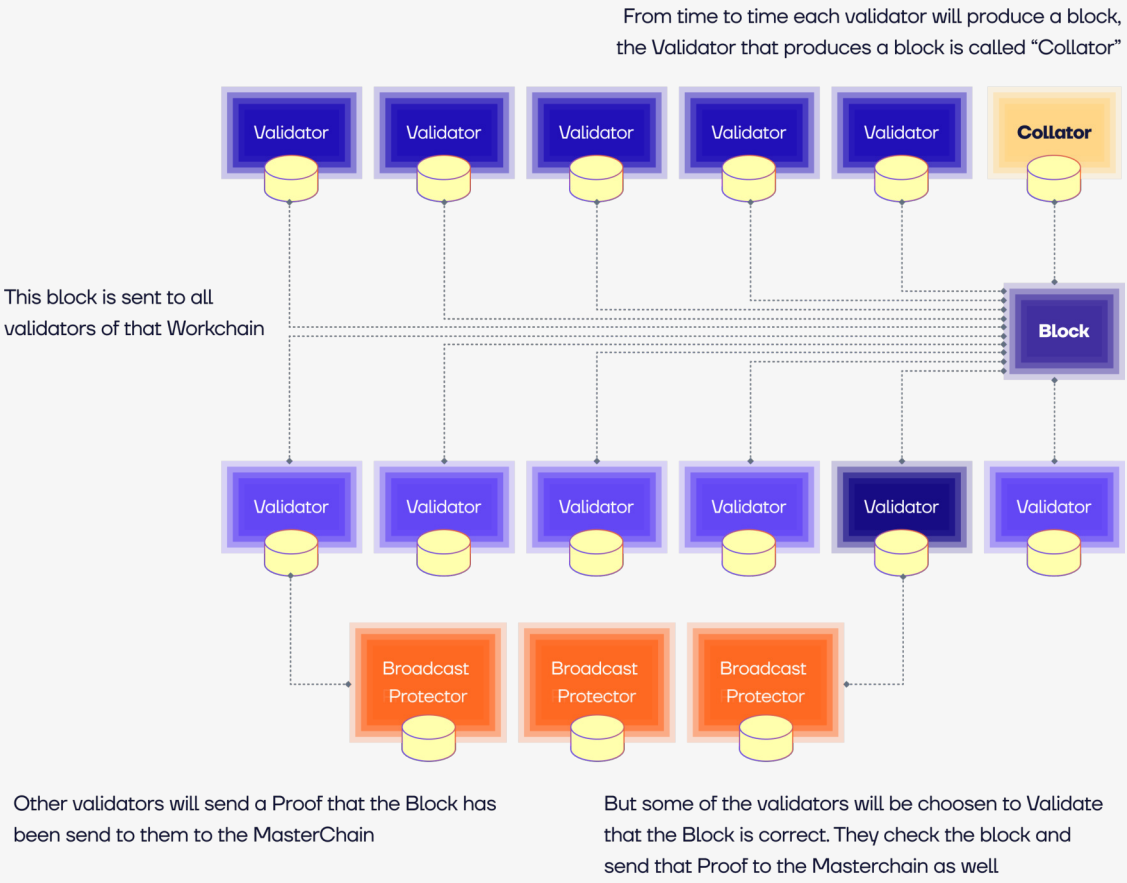
Now let's talk about security guarantees. If there is one chain where all the validators validate everything, then obviously the security guarantee for this network is the combined stake of all of its validators. Basically a lot of validators equals a lot of security guarantees. If, however, we start dividing them by segments such guarantees decrease.

And if these segments are split into even smaller ones, the security guarantees decrease to unsustainable levels, which means that corrupting the block becomes financially viable for scammers and rogue validators. The corruption will go undetected for some time because other validators will not run this computation. So they won't know a scam has taken place.

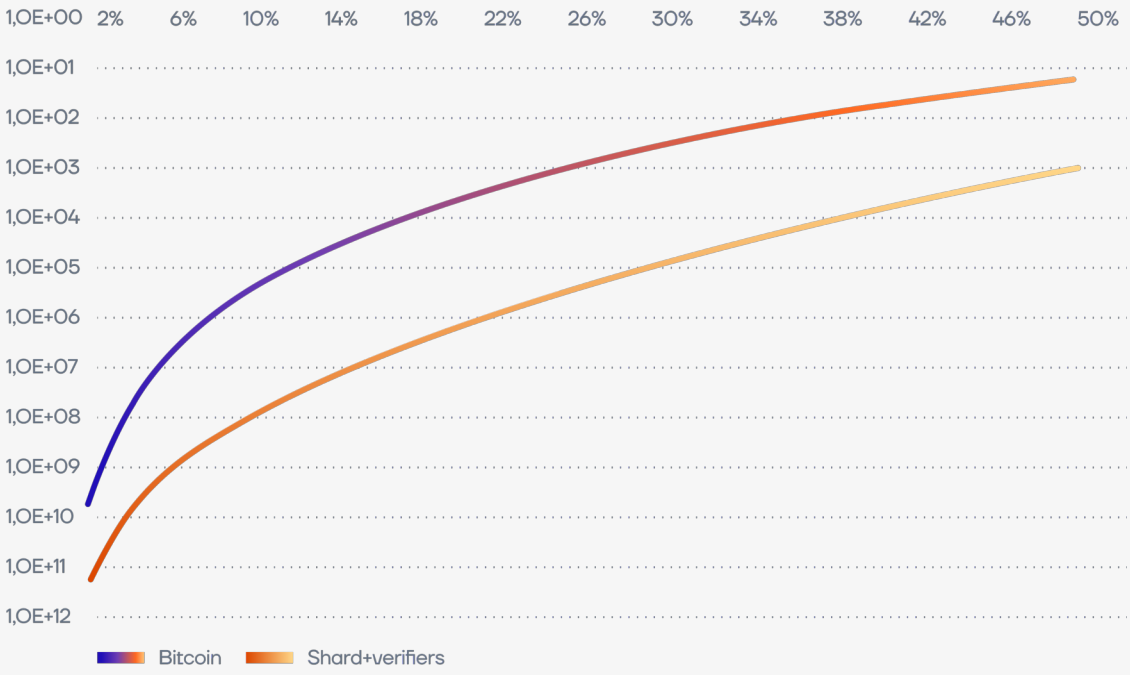
The proposed way to deal with this was — let's have fishermen.

What fishermen do is they validate some blocks. And then one of them may detect that the wrong block was produced, which they can actually prove. With this proof, they can get some reward and the network can slash the validators. But there was a problem with that approach. When the fisherman finds this wrong block, it may be just too late. The rogue validators can submit a very large fake transaction, and then do something with that transaction before anyone has noticed. So for example, they can double spend: create tokens out of thin air, send this money to some smart contract of some exchange and sell the tokens to those of another network, before running off with the money. And they can do all that before fishermen will ever know and catch the block. This is because fishermen are probabilistic. They do not validate every block; submitting the proof and slashing the validator will take time and will not reverse everything the validator did. And the scam will, of course, be much more valuable than all the combined stakes of all validators that will be slashed later on. And that can have catastrophic consequences on the network. Simply put, we can not rely on fishermen. And it doesn't matter if the blocks are corrected afterwards, because someone already suffered a lot of losses, for example, the exchange or the bridge.

So what can we do in order to verify any wrong block before it will ever be submitted to the network? In Everscale there is a protocol we call “Soft majority fault tolerance” SMFT for short. And we'll describe it very simply now.



A Validator who proposes a new block is called a Collator. The Collator will produce a block and send it not only to thread validators but to all validators. And that's normal because we remember the data here is the same. So now all of the validators have all of the blocks, even if they don't all validate them. Then, through the wonders of algorithmic science, each and every validator will run a calculation on that block hash. Through this, a few out of all validators will now need to validate this block in addition to those traditionally in their purview. We call these validators — Verifiers. The function that calculates who the verifier is, is random. The collator can never know which of the other validators will be verifiers. As a result, they can neither cheat, nor collude because they will need to collude with at least 51% of the network. And if 51% of the network agree to collude, it's no longer an attack. It's the correct network. If 49% think that the block is wrong and 51% think that this block is correct: this block is correct by definition. Therefore the chances of the collator to predict the identity of the verifier (even if the collator is in cahoots with 50% minus one validator of the network) are so small, the attack is never probabilistically successful. Thus the collator will never propose a wrong block because the collator's chances of succeeding with this attack is mathematically lower than in trying to corrupt Bitcoin.



Not magic: because it is impossible to know which validator will be chosen to verify the block and because there was a proof that the block was indeed sent to everybody on the network, the Collator will never try to send wrong block because their chances to succesfully lie about it are less than that of in Bitcoin after 6 blocks confirmation

Of course there's one small problem: we need to be sure that this block has, in fact, been transmitted to all the validators. In order to ensure that there is another protocol, which proves the block propagation and it's called Broadcast Protection Protocol. This protocol ensures that we can prove that a block has been transmitted to at least 51% of all the validators. Once we know that, we know that the attacker can never be able to predict, with a high enough chance, who the verifier is within 51% of the validators. If the Verifier finds a rogue block there is a procedure on how this block will be stopped and checked by a lot of other validators to be really sure that this block is wrong or correct. Now, we also ensure that post factum we know who the verifiers of the previous blocks are, and if the verifiers did not perform their job will slash them as well. So they will be penalized if they didn't verify the block. So, the collator cannot also assume that verifiers will be lazy enough to not validate blocks.

Chapter IV. Reliable External Messaging Protocol

When we post a picture on Facebook we don't expect the process to take minutes, or even half a minute. We, more or less, expect it to happen instantly; two, three seconds, after that our patience wanes. However, in blockchains today, when a transaction is sent, the wait is long; half a minute, many minutes etc. We want to ensure that the user has the same experience working with the blockchain as with other modern IT systems like Facebook or Instagram. The problem, of course, is that when users send the message, it sends a message to some smart contract, which then needs to be executed, and the validators need to validate the execution. This process takes time. How can we decrease the time that it takes users to see the result of the execution of their message on the blockchain?

In answering that we must address the other problems which need to be tackled in this paradigm: replay attacks, DDOS and front running attacks. A replay attack is when the smart contract is fed the same messages, to execute the same transaction, several times, instead of only once. Protection against the replay attack now needs to happen inside the smart contract, which takes execution time, which takes Gas and so on. And then there is a DDoS, this is when the users can just load the network with many external messages, millions of messages, for free, and make a server halt. This, incidentally, happens often on blockchains. Front running attacks are where, for example, a trade has to be made and a message is sent with a trade order, someone else sees it, and they send another message, which then reorders into being higher than the original message and it can take advantage of it.

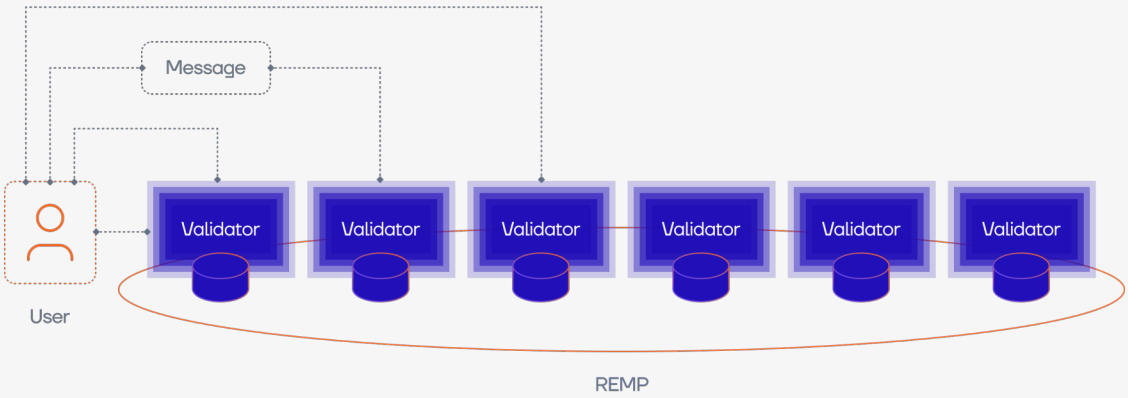
REMP solves these problems.

If a REMP validator accepts a message and sends an acceptance confirmation to the user, then this message will most certainly be included into the block. The validator then sends status about what happened to the message to the user along the road. The message will be included into the block in the order it was received by the validator, meaning that, even if the block collation failed, it will be included into the next block in the same order. This is protection against front running attacks. And it also ensures it will be included only once which is a protection against Replay attack.

As was previously explained, the collated block has an extremely low chance of being wrong, meaning that the user can assume that once the acknowledged message from the validator is received via REMP, the message will be included in the block; that the transaction is executed correctly and the user can close the application. In terms of the User Experience that means a sub-second finality (the message to and from the REMP takes around 50-100 ms., while the REMP processing takes another 500 ms. on average. All in all, less than a second.)

REMP also has a built-in DDOS protection which ensures that if the user sends a lot of messages, which are failing on execution then this user that sends those messages will be banned for ever increasing periods of time.

So again, REMP protocol ensures sub-second finality, replay protection, DDoS protection, and front running protection.

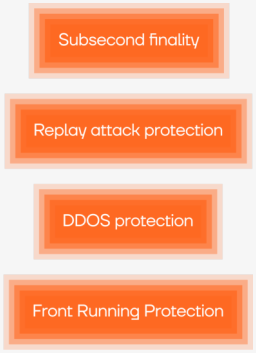


External Message

Users sends a message to smart contract it wants to be executed to all validators of a Ever OS thread responsible for the execution of that smart contract at that time. Message is proccessed by a special protocol called REMP to ensure that the message is certainly included in the block if accepted and executed once one.

Because as described earlier once the block is collated there is an extremely low chance it is wrong, the user can assume that if the message is accepted into REMP it will most probably be executed. For most parts this ensureas subsecond message execution finality.

REMP has built in DDOS protection mechanism ensuring the blockchain is never spammed with too many empty messages.

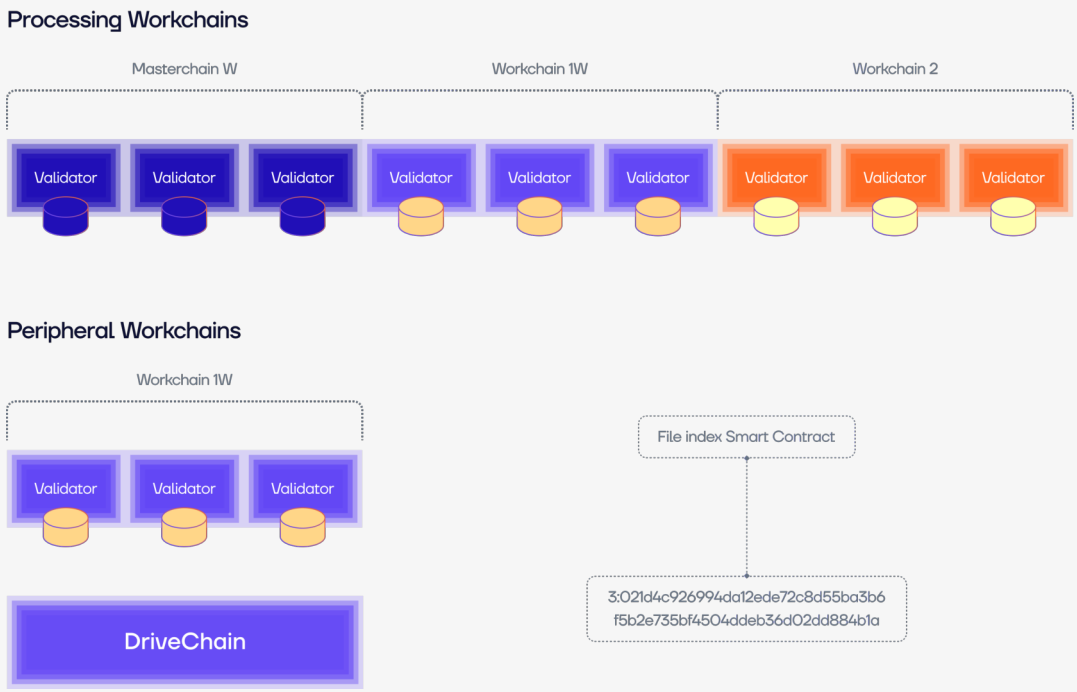


Chapter V.
Peripheral Workchains.

We've already established that WorkChains scale the network when more processing power is needed. These WorkChains are called Processing WorkChains. However, there are also Peripheral WorkChains. They operate like the peripheral devices that you connect to your computer, for example, a printer or a hard drive. They're like resources that your computer can use and support. Ever OS is no different in that regard.

Let's consider one example, a DriveChain. This is a decentralized storage device for Ever OS. It is a chain optimized for storage of large objects. DriveChains have special smart contracts residing on them. In order to store a file in the DriveChain, one needs to deploy the file index smart contract (like an index used in Unix or Linux operating systems), which contains certain information allowing for storage and for the retrieval of files and pay for that storage.

We can imagine multitudes of different Peripheral WorkChains. Perhaps ones specially optimized for really long-term or forever storage of files. Anyone can launch a Peripheral WorkChain at any time, while providing economic incentives for validators to join. It can be used exactly like plugging your devices in any modern operating system.



Chapter VI. Distributed programming

What is this distributed programming we've heard so much about? On blockchains today wallet addresses are associated with a public key, derived from a private key. The address can contain a smart contract code or not contain anything at all and just be used for a native token.

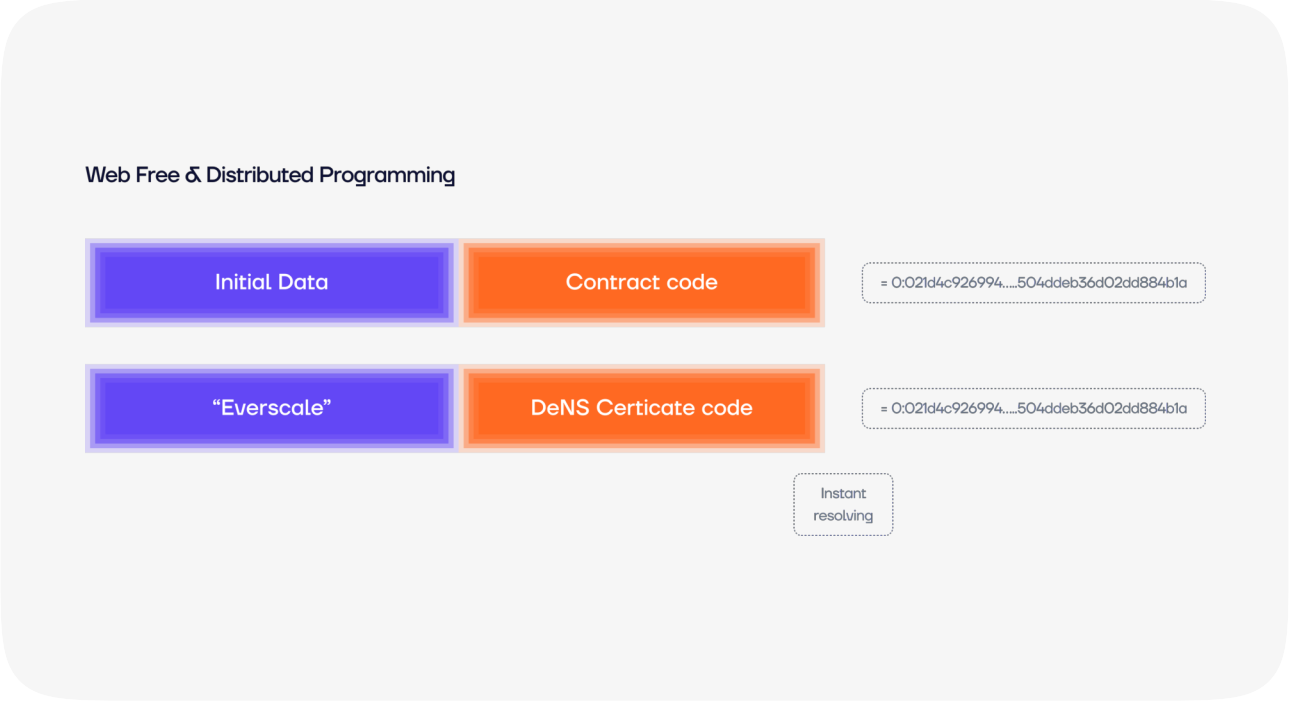
On Everscale, however, the address is not associated directly with a public key, and must instead be associated with a smart contract in order for someone to be able to use it. Money can still be sent to any address, but those tokens will just sit there without anyone having any possibility of doing anything with them until smart contract code, a program that runs this money, is attached.

In that sense, Everscale is a definitive smart contract platform because every address is a smart contract. But more interestingly, this address is a result of a calculation which hashes the initial data and the code of a smart contract. For example, inputting the public key along with the contract code of a wallet, will output an address. However, if a different wallet's smart contract code is input, then the output will be different. There are therefore infinite amounts of addresses that can be derived for a single public key.

This opens very interesting possibilities, and has many positive implications. Let us take an example of how filenames run on Everscale.

Let’s say we want to resolve the name Everscale inside the Everscale blockchain. For this we use the decentralized name service certificate smart contract (DeNS). This is a smart contract associated with a certificate service, used to resolve addresses, similar to how today's browsers work. However in this case it just needs to be downloaded once, and can be used to resolve locally any name in the domain, without a need for a server. All it takes is for this certificate to be opened in a browser, the word ‘everscale’ to be typed and, because the address is just a function of instant hashing, in fractions of a second, the address appears, having been calculated on a machine locally.

This is the best decentralized name service in use today, entirely operating without servers. When we talk about distributed computing, we think of Everscale as a sort of a distributed decentralized key value database.



Chapter VII.
Gas & fees

We're moving now to the question of Gas and fees, as well as, more generally, economics. There are two types of fees. Storage fees and processing fees, called Gas. While there are pretty complicated fee structures in Everscale, we're not going to talk about that in all too much detail. What is important here is how much users pay.

We know that with the number of transactions in the network increasing, the Gas fees increase as well. This is because of network congestion. This potential for congestion means fees cannot be arbitrarily low, to protect the network from an attack where someone buys it and stops it entirely for a low price. So in order to have really low transaction fees, the network has to be really scalable. That's just a matter of arithmetic. We have such a network. So let us discuss how the fees are going to be structured.

Let's take a MasterChain with a WorkChain on top of it, which has threads in it. This WorkChain has a current network capacity of around 10,000-12,000 transactions per second. In the beginning, the gas fees are naturally going to be very low because the volume of transactions offsets any potential congestion. However as the network grows, and starts becoming congested, Gas prices begin to increase. Why? Because we are already anticipating that we need new validators for new WorkChains that we need to set up. In order to do that, we need to start collecting fees in order to incentivize new validators to join the network.