# Xenophyte - Fast, Lightweight, Portable, Decentralized Cryptocurrency

Development start
date: 06/09/2020
Date: 14/10/2021
Last updated: 28/07/2025

**→ Titles:**

## Summary:

**Xenophyte** is a technology developed mainly in C#, it allows to easily create, understand a decentralized blockchain process, it can be configured to support a sovereign update system, allowing to include updates on it without editing the Blockchain content. It has been developed to update **Xenophyte**, which is a centralized cryptocurrency.

This system allows to certify a public synchronization node as trusted, to update the mining process method without the need to have a fixed and centralized network point, the updates are exchanged between the nodes, these updates have a block height of activation, allowing to ensure the broadcast of them more easily before their activations by the nodes/users who have them in memory.

The method of mining process **PoW&C** (Proof of Work and Compatibility), is based on different reproducible processes according to the data emitted by the miner, this process uses information from the previous block to the one currently targeted but also a part of the block of this one.
The content of this work holds the identity of the miner's portfolio, allowing to avoid the absorption of the work by an edited synchronization node. **An edit of the job, radically changes the final value of the job, which prevents a third party from appropriating it.**

The transactions sent between users are obviously signed by their private keys, their representations are signed, the representations respect a precise structure to simplify, accelerate their research if necessary.
This technology, has a memory management system, allowing not to keep the entire blockchain in memory, to work directly if necessary with the cache system, but also to do **High Scalling** of data across different machines, in order to obtain a system fast and stable enough and functional on the long term.

The data synchronized from the public nodes, are always those issued identically across the majority of nodes contacted, obviously a grid of internal checks to ensure that all data retrieved are functional and always compatible with previous data held by the user / node of the **blockchain.**

only the nodes certified by sovereign update as being those having authority to deliver synchronization data.

Here is a list of some technical details, as well as the default configuration that you can find in the source codes of this project, so you can make your modifications at your convenience:

## Technical details:

→ Development language: C#
→ Framework: **Net 5 (Backwards compatibility: NetFramework 4.8)**
→ CrossPlatform: Yes via DotNetCore
→ Support: Windows/Linux/BSD, ARM (Raspberry PI 3 tested)
→ Signature technology used: ECDSA with SHA3-512, Curve: SECT571R1
→ Blockchain version: 01

Note: The **SHA3-512** hashing method *is used in the set of processes that need it to minimize potential* ***length extension attacks****,* ***SHA2-512*** *is more venerable, although some significant power is required to attempt this kind of attack.*

## Default configuration:

→ Cryptocurrency name (default): **Xenophyte**
→ Cypher: **XENOP**
→ Address format: Base58
→ Base58 encoding checksum size: 16
→ WIF wallet address length: 109
→ WIF public address length: 219
→ WIF private address length: 119
→ Maximum number of decimal places: 8
→ Maximum number of corners that can be distributed: 26,000,000.00000000
→ Cost of a minimum transaction fee: 0.000002
→ Cost per kb relative to transaction size: 0.000002
→ Number of confirmations of a block reward: 10
→ Minimum number of confirmations per transaction: 2
→ Default minimum number of target height blocks: 5
→ Maximum number of transactions per block: 100,000
→ Sample blocks used to gauge the value of a transaction fee: 1440 blocks
→ Dev Fee: enabled
→ Dev Fee percentage: 0.005
→ Block reward amount: 10.00000000 - Dev Fee if enabled.
→ Coin Halving: enabled
→ Coin Halving block interval: every 100,000 blocks unlocked
→ Sovereign Update Support: enabled.
→ Block Time: 60 seconds.
→ Expected number of blocks unlocked per day: 1440.
→ Sample blocks used to calculate the next difficulty: All the previous 480 blocks.
→ Decimal accuracy of difficulty calculation: 100000
→ Minimum difficulty: 1000
→ Pre-mined default amount (Genesis Block): 2,973,370.00000000
→ Number of Genesis Block transactions: 1
→ Number of confirmations per network of a confirmed block: 2
→ Maximum time of a share unlocking a block: +/- 60 seconds
→ Network marking key: enabled, allows differentiation between different networks by changing it.

# I. Description of the PoW&C mining process

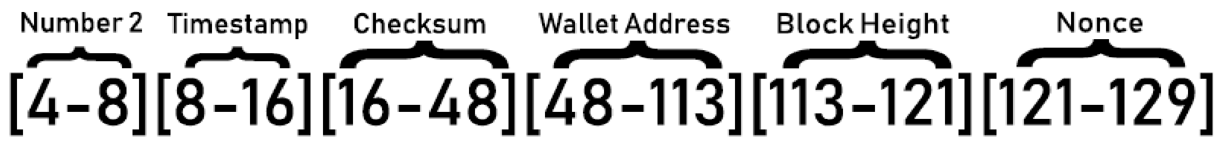
→ **Summary:**

## Summary:

**PoW&C - Proof of Work and Compatibility**, is a mixture of PoW (Proof of Work) and a compatibility proof.
By reusing a part of the previous block to the target one, and using information from the target block, a proof of compatibility is thus submitted to the proof of work that follows the process, this allows to ignore any inconsistencies, incompatibilities by invalid shares more easily.
Part of the content of the share must contain the wallet address of the decoded miner, circumventing potential problems of work absorption by a public synchronization node that has been edited and would like to recover the work done.
**Any changes in the content, nonce, encryption, but also on the data used directly change the final value of a share.**
It is therefore normal that a share is no longer functional on a new target block and if edited, that its final value is no longer the same as that of the original work.

## 1. Contents of a share.

A miner first generates a PoC content (proof of compatibility) with random parts inside. The size of this content must be equal to 129 bytes. Any differences will be considered invalid.



PoC structure of a share :

:

*This diagram shows the data structure of the content of a share, specifically the generated PoC.*

→ [0-4][0-8] = First Number | Second Number = the number of transactions in the previous block.
→ [8-16] = The timestamp of the share.
→ [16-48] = Random checksum data, if the miner uses a Pool, the Pool communicates it.
→ [48-113] = Decoded wallet address, if the miner uses a Pool, it is the Pool address that will be communicated and used.
→ [113-121] = The height of the target block.
→ [121-129] = The nonce used.
Total: 129 Bytes, representing the total contents of the PoC.

**Specifications:**

The numbers contained in the PoC, are used to return the number of transactions from the previous block, only certain mathematical operators are used: + * % -
The timestamp contained in the PoC, is used in the case that the final value of the share is sufficient to unlock the target block, is close enough to the Block Time, to accept it, it is appropriate that in the established processes, that any cheating of the timestamp, is not tolerated.
For example, a timestamp much too early compared to the current one, will not be valid, as for a timestamp much too late.
Also the timestamp contained must be indicated in the network request used for the broadcast, in the case of absolute validity, the target node(s) will execute a broadcast of the work provided targeting the other nodes.

## 2. Processes performed on a selected nonce.

When the PoC is generated correctly, the miner selects a number called Nonce, between a minimum and maximum range accepted in the mining process configuration.
This generated nonce, reworked through different processes, is used in the final method of encrypting the share data as an IV.
When a synchronization node is called to check a share in order to validate it and unlock the target block as well as to perform a broadcast, all processes dedicated to this part are used to check if the source nonce returns the generated IV.

Default configuration of a valid nonce:

→ Minimum: 1
→ Maximum: 4294967295

1. First the process generates the nonce between valid interval, via a secure random generation method.
2. The generated number is converted to an array of bytes to be thus worked with the following processes.
3. A number precisely established in the mining configuration of SHA3-512 computations are applied on the nonce.
4. Then, an XOR process makes changes on the result of the computations.
5. A method, generates points, returning angles, allowing to generate a virtual pseudo square, a certain number of computations are carried out, if a square is found, the process ignores the other computations to be carried out, if not another number of SHA3-512 computations will be carried out, this process gives a dose of random "chance".

6. The first two processes of computations are performed again.
7. Compression of the work through the LZ4 compression algorithm.
8. Finally, generation of the encryption IV by derivation RFC2898.

## 3. Encryption process performed from the share.

To encrypt the work of the miner, the representation of the transactions of the previous block (final transaction hash) is used as key of encryption, to complete the encryption the IV generated through the nonce selected by means of the processes of performed work is used.

To generate the encryption key, the representation of the transactions is subjected to a SHA3-512 computation, the result obtained returns an array of 64 bytes it is thus resized to the size of 32 bytes to be functional as an encryption key.
*The proposed mining tool is optimized to keep this generated key in memory until the target block is still unlocked.*

AES configuration:

→ Key size: 256bits.
→ Block size: 128bits.
→ Mode: CFB.
→ Padding: PCKS7.

A number of established in the mining configuration of encryption loops are performed.
When this task is performed, a data converted to hexadecimal share is equal to **352 characters**. This data is important for the calculation of the value of the generated share.

## 4. Share value calculation.

The process of calculating the value of a share is simple, a SHA3-512 computation is performed on the encrypted share, the generated representation is then converted to BigInteger. Finally a mathematical calculation between the difficulty of the target block, as well as the maximum possibility of representation of a SHA 512 can offer is performed.

Mathematical formula:

share_value = BigInteger(SHA3-512(share_crypted))

final_share_value = (2^512 / block_difficulty) / (share_value / block_difficulty).
Example, assuming the share value is approximately equal to 2^494 and the target block difficulty is 300000, the final calculation is:

( 2^512 / 300000 ) / ( 2^494 / 300000) = 262144

In this case, the value of the share is lower than the difficulty of the block, so the share will not be accepted to unlock it.
If it is higher or equal to the difficulty of the block, then it will unlock the current block.

## <u>5. Verifications performed on the share.</u>

A verification carried out on a share is carried out in the same way as the generation of a share, except that this one is carried out in reverse, with various methods beforehand to verify its contents as well on the request received as on the contents of this one, as well on the format of the contents, length for example.

As previously explained, the PoC timestamp contained in the encrypted share must be identical to the request sent by the miner.
It is the same as the wallet address provided in this request, it must be identical to that contained in the PoC.

The request must indicate exactly some information of the target block, its height, its hexadecimal representation (block hash).

The request must also indicate the nonce used, the hexadecimal representation of the generated IV, the encrypted share in hexadecimal format, as well as the value of the share that must be retrieved during verification.

<u>- Structure of a share sent in JSON format:</u>

```
{
        WalletAddress: "wallet_address",
        BlockHeight: "target_block_height",
        BlockHash: "target_block_representation",
        PoCShare: "the_share_crypt_in_hexadecimal",
        Nonce: "the_selected_announcement",
        NonceComputedHexString: "the_iv_generated_from_the_advertisement",
        PocShareDifficulty: "the_value_of_the_share_computed",
        Timestamp: "the_timestamp_content_in_the_share_poc"
}
```

<u>- Receipt:</u>

When a share is received, a node will only accept shares that have a timestamp that is recent enough or not too far behind to consider it.

→ Checks performed:

1. Checked the format of the portfolio address contained in the request.

2. Verification of the block hash, to ensure that it is identical to the targeted one and valid.

3. Verification of the selected nonce.

4. Verification of the content of the provided encrypted share, empty, invalid format/length for example.

5. Verification of the share value according to the provided encrypted share.

6. Verification by decryption of the encrypted share.

7. Verification of the supplied IV by generating through the supplied nonce that it is the same.

8. Verification of the compatibility proof.

9. Verification of the timestamp contained in the PoC.

10. Comparison of the share value as valid to unlock the target block.

→ Broadcast of the share:

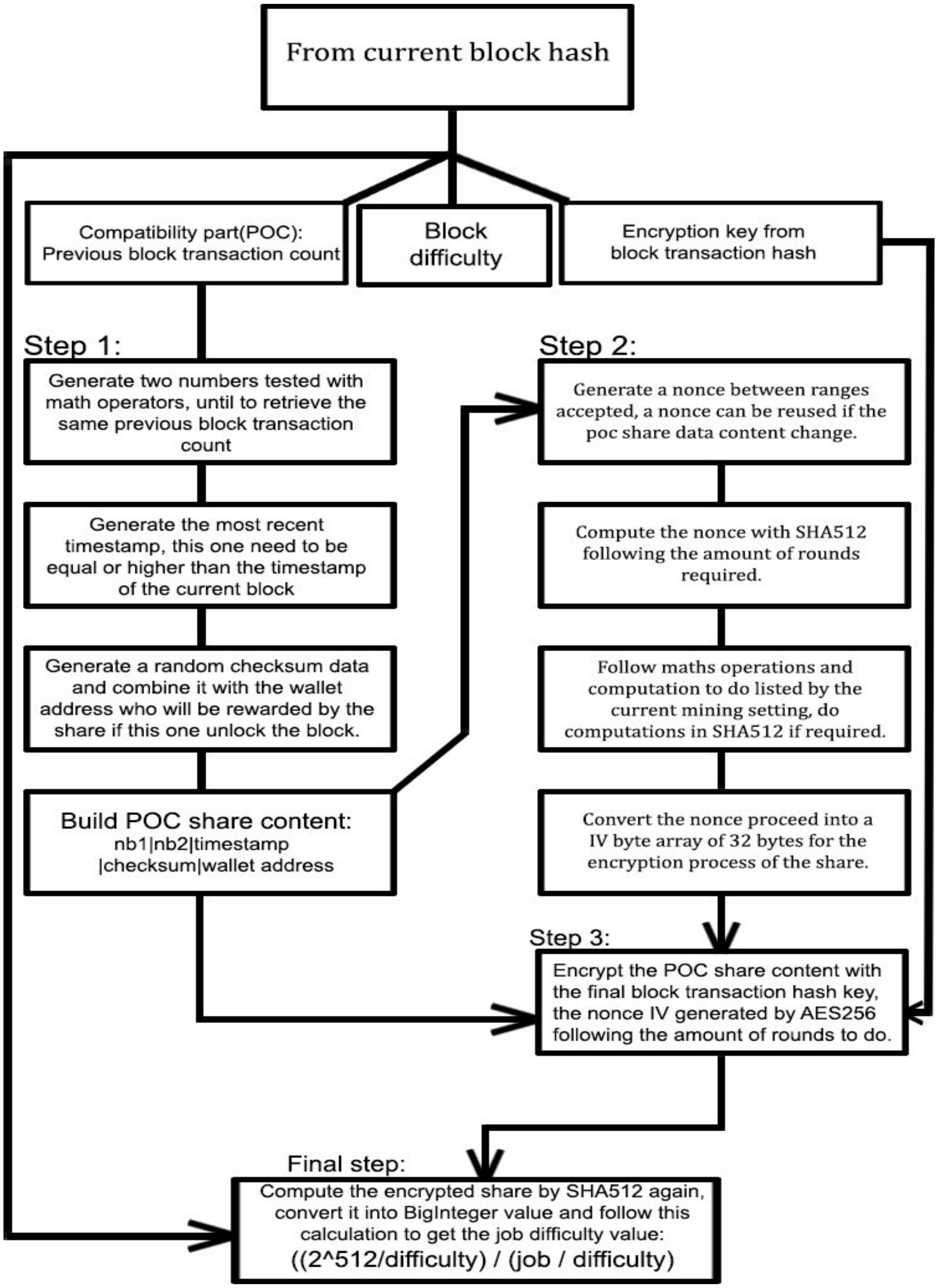
When the share provide unlocks the target block, the targeted node(s) performs a broadcast of it to all nodes listed in their memory to ensure that it is taken into account by the majority of nodes in the network.

→ Network confirmation step:

This step is performed in the nodes synchronization system, the principle is simple, called the different nodes in memory to return the metadata of the unlocked block,
if everything is identical a network confirmation counter is incremented in the memory of the node, when the limit is reached, it is considered available in the majority of the decentralized network.

## 1. Diagram of the mining process.

In an attempt to simplify all the information, here is a diagram representing the processes carried out.


From current block hash
Compatibility part(POC): Previous block transaction count
Block difficulty
Encryption key from block transaction hash
Step 1:
Generate two numbers tested with math operators, until to retrieve the same previous block transaction count
Generate the most recent timestamp, this one need to be equal or higher than the timestamp of the current block
Generate a random checksum data and combine it with the wallet address who will be rewarded by the share if this one unlock the block.
Build POC share content: nb1|nb2|timestamp |checksum|wallet address
Step 2:
Generate a nonce between ranges accepted, a nonce can be reused if the poc share data content change.
Compute the nonce with SHA512 following the amount of rounds required.
Follow maths operations and computation to do listed by the current mining setting, do computations in SHA512 if required.
Convert the nonce proceed into a IV byte array of 32 bytes for the encryption process of the share.
Step 3:
Encrypt the POC share content with the final block transaction hash key, the nonce IV generated by AES256 following the amount of rounds to do.
Final step:
Compute the encrypted share by SHA512 again, convert it into BigInteger value and follow this calculation to get the job difficulty value:
((2^512/difficulty) / (job / difficulty)

## I. Description of the memory management system.

To give the possibility to remain functional on the medium/long term, a cache system has been designed for this problem, allowing the nodes but also the Desktop Wallet of the users to function in time whatever the data load that the synchronized Blockchain imposes.

This cache system is designed to work only with what is necessary to have, in order to avoid any problems, loss of performance, compatibility, and others, no external technology has been used to design it, this allows to have an independent system, working only with the established principles and processes.

By default the cache system is configured in **DISK** mode, a certain number of file indexes will then be created according to the blockchain.

The streams are maintained, making the execution of a read/write instruction faster, the streams are also opened in asynchronous mode with random access, giving an additional indication to the system for caching changes on the files.

Each updated item is written to the end of its file index, so its total length and position is saved in memory so that it can be quickly re-read on demand without having to re-read the entire file index dedicated to cached data.

An asynchronous task in the memory management is executed over a reconfigurable time interval to rewrite the contents of the cache files in order to save disk space.

When an element is called several times over a short time interval that is also reconfigurable, it is stored in memory if the maximum memory allocated allows it.

When the maximum memory allocation is reached, the cache system will attempt to save the updated memory data to the file indexes and remove it from memory, in order to obtain enough available memory to hold the item.

If the item in question cannot be stored in memory, it will be written back to the cache if it has been updated, otherwise it will simply be ignored.

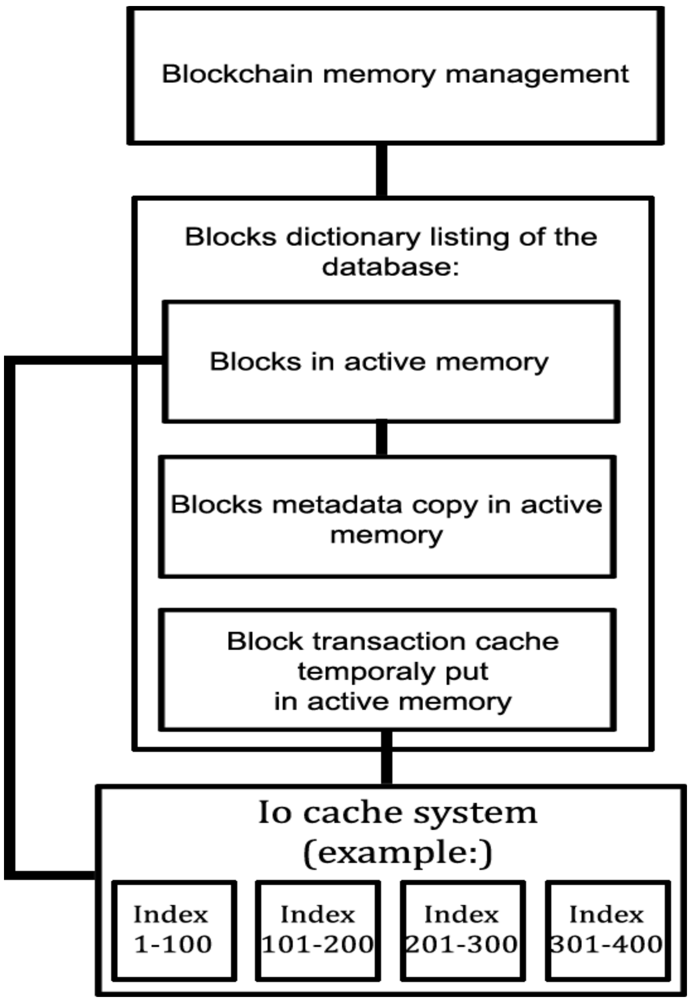
Items deleted from active memory that have not been updated will also be ignored, and thus will not be rewritten since their current versions are already written to their cache file indexes.

To ensure the operation of this system, each index has a Semaphore, allowing to block multithreading access when a task is already in progress, it will be released when the task is completed, cancelled.

A file index can contain several synchronized blocks, this number is also reconfigurable in the cache system configuration.

For the **NETWORK** mode of this system, all the processes described are also carried out, but with the difference, on the host or network hosts that hold the data, a network stream is then opened to execute the various orders equivalent to the system in DISK mode.

Finally, here is a simplified diagram of the Blockchain's memory management system:


Blockchain memory management
Blocks dictionary listing of the database:
Blocks in active memory
Blocks metadata copy in active memory
Block transaction cache temporaly put in active memory
Io cache system (example:)
Index 1-100
Index 101-200
Index 201-300
Index 301-400

*A little schema of the blockchain memory management with the Io Cache system.*

**1. Reading:**

Once it's necessary to retrieve back a block or a transaction, the system check if the block data is still in the active memory and return it if it's the case.

Otherwise, the IO Cache disk will read by the last position registered on the io cache file and by the last length registered.

Depending the setting, the IO Cache disk can keep alive pending a certain amount of time, the block data readed.

On some cases, if for example, the max amount of active memory dedicated to use on the IO Cache System is reach, the system will try to write enough blocks data who are still in the active memory into io cache files, to finally get enough memory for read and return propertly the block data asked.

The read process of a block data, will try to minimize at maximum the CPU/Disk usage, by reading the block data per block size, depending his size, the read by block of data, will generate a percent of block data size, and read until the length of this one is reach.

## 2. **Writing:**

The IO Cache system, write new blocks data push inside it, this one also write a block data updated, but only on some cases:

1. If the block data is newer, the io cache system, write directly at the end of the IO Cache file dedicated to the index linked of the block height, without to keep it in the active memory at first. The writing process save the position of this one and his length.

2. Once a block is updated, depending some cases, the block updated is keep inside of the active memory. If not the block data updated is written at the end of the IO Cache file dedicated and save the last position and the length of this one.

3. Finally to save a maximum of disk space, a scheduled task rewrite all last versions of each blocks data on a backup file, once it's done the backup file replace the original IO Cache file, this purge task is done a specific scheduled time internal and on a specific case.
This task also get out blocks written, of the active memory, but depending some asking interval, some of then can stay on the active memory.

The whole purge of a IO Cache file need to reach a certain percent of writting data, this percent can be edited on the setting of the node tool or of another tool who support this sytem.

4. To save disk/cpu usage, like the reading system, a block data is written by block of data, divided by a percent of size calculated from the block data size itself.

## 1. **Memory management:**

Every blocks registered on the IO Cache system, have their amount of memory saved, the calculation is different, becauses objects, variables inside the active memory provide another cost.

This memory management permit to identify if the amount of memory dedicated to the system is reach or not, by asking the main system to make a sum of active memory of blocks who are still in the active memory, but instead to recalculate each amount of memory used by each blocks, the amount of memory used by a block is recalculated at each insert/update of them. This way permit to save time, and cpu power.

### 2. Potential updates:

Some specific strategies will be implemented with the time on this sytem, to permit to not read whole block data, for example if a simple transaction is asked, this one can be retrieved across the reading of a block data, without to read it completly. And then return back this one.
Those kind of strategies require more work and tests, and should be implemented later, to increase scability and stability for the long term.

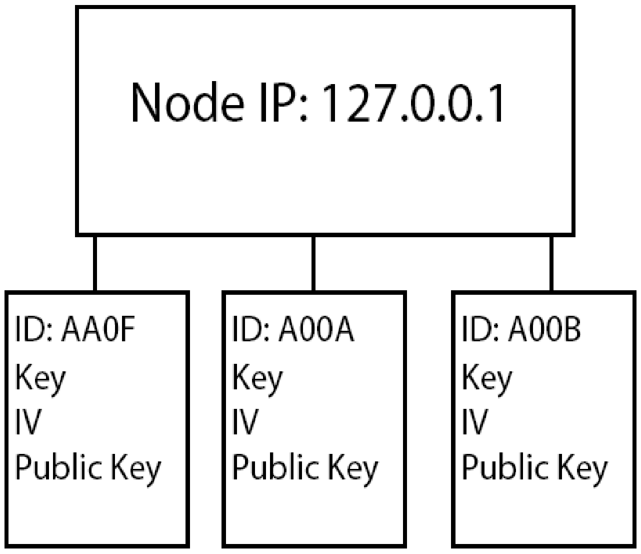
## II. Description of the P2P communication system between nodes.

### 1. Structure of a listed node:

Listed nodes are listed by IP but also by a UniqueID generated and communicated by the node that exchanged different information.
This listing system allows to list several public nodes with different information, but also network configuration, such as the communication port open to the public.
By giving this possibility, a public node host can easily stop one of them while keeping at least one or more nodes accessible.

The encryption keys and the public key used to verify the signature of the sent/received packets can be identical from one node to another or completely different, by default a node contacting another node will generate different keys for the communication.
The encryption of communications is not really necessary but brings an additional step of compatibility through the whole decentralized network.

Here is a simple listing diagram of several nodes all having the same public IP:



*Default configuration of a node:*

→ Unique ID hash length: 128 (Not configurable)
→ Default P2P port: 2400
→ Minimum P2P port: 1
→ Max P2P port: 65535
→ Max P2P connections per IP: 1000
→ Max API connections per IP: 1000
→ Semaphore max timeout per entry from the same IP: 5 seconds.
→ Synchronization per row: Enabled.
→ Max synchronized transactions per row: 5
→ Max synchronized blocks per row: 5

Other options are configurable, and allow to optimize, adjust the network configuration of a node.

## <u>2. Description of the synchronization system:</u>

The synchronization system of a node executes several asynchronous tasks in a synchronization instance, they are divided into several types of tasks, allowing not to perform this one by one.

Each synchronization task executed has a list of nodes generated, updated, and keeps the connections with them open as long as possible to speed up communications.

The system of updating the lists of nodes, can also add new nodes if in short course, new ones were recovered, and to remove those which would be inaccessible, or having returned too many invalid elements.

### <u>Synchronization task list:</u>

→ <u>Node list request task:</u>

This task requests potentially unknown nodes from the nodes already listed in memory, allowing to expand the number of known contact nodes.

→ <u>Sovereign update synchronization task:</u>

This task asks known nodes if they have sovereign updates, they are retrieved, checked and installed if they are valid and previously non-existent on the node.

→ <u>Blockchain progress synchronization task:</u>

This task asks the known nodes, the current progress of the Blockchain, the majority returning the same progress, will be taken into account.
An internal check is performed to ensure that the data is valid.

→ Block synchronization task:

This task asks the known nodes according to the current progress of the Blockchain retrieved in the synchronization task dedicated to it, the block metadata.

Depending on the configuration of the node, one or more block metadata are retrieved.
The recovered metadata are those being returned identically via the majority, internal checks are performed before taking them into account.

Once the recovery of the metadata is done, one or more tasks of synchronization of their transactions are done, in the same way, the data used are those issued by the majority and internal checks are done.
To finalize, the synchronization of a block, the final transaction hash of the block recovered by synchronization of the metadata of the block in question is compared with the hash representation of the synchronized transactions, if this representation is identical and all the previous checks are valid, the node believes that the block is valid.

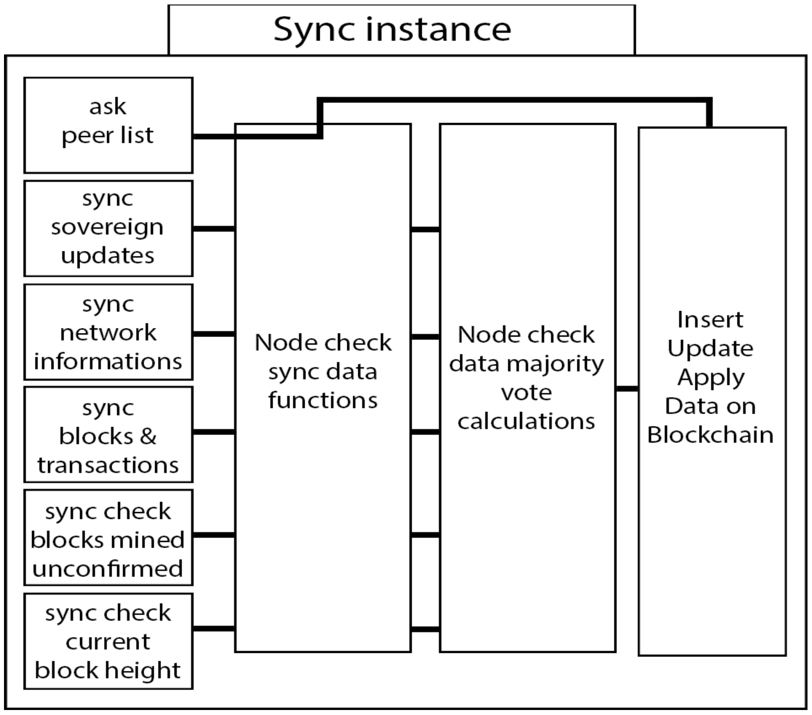
→ Unconfirmed block verification task by networks:

This task asks the known nodes based on the blocks contained by the node, to check if the majority accurately returns the same metadata of the unlocked blocks that were not confirmed by networks.

If one or more blocks are not identical to the majority, they are resynchronized.
Once this step is done, the network-confirmed blocks can be considered in the transaction confirmation task that is performed internally.

→ Current height block status check task:

This task asks the known nodes if the current height block is unlocked, this task assists non-public nodes or public nodes that could not successfully receive the broadcast of a share unlocking it.
This check always takes into account the result of the majority and internal checks are performed.


Sync instance
ask peer list
sync sovereign updates
sync network informations
sync blocks & transactions
sync check blocks mined unconfirmed
sync check current block height
Node check sync data functions
Node check data majority vote calculations
Insert Update Apply Data on Blockchain

Here is a simple diagram describing the process of a node's synchronization instance:

The packets sent/received are in **JSON** format and formatted in **Base64** separated by a unique separator character to ensure their transmission. The separator character indicates the end of a packet, if during reception the sum of the data exceeds a certain limit, all the received data will be deleted and the reception task will be cancelled.
As previously stated, communications between nodes are encrypted, so it is the case during synchronization tasks.
Each open connection has a connection object to send a request and receive the expected data, this ensures that each request is received in parallel.

## 3. Description of the broadcast system:

The broadcast system is only used to transmit transactions, mining shares to other public nodes, which themselves will broadcast if the transmitted data is new and valid.
These tasks are done asynchronously and therefore independently of all other tasks performed by other systems in a synchronization node.

A **Desktop Wallet** or an **RPC Wallet** also broadcasts the data they send, however by default they are not available to the public to receive data from outside through this system.
They can synchronize transactions waiting to be picked up by a block from public nodes.

Two broadcast tasks are dedicated to the transactions:

Two lists of instances are generated continuously, as long as nodes are available, these instances are then executed and their states monitored.

The instances open connections to the targeted nodes, while trying to keep them open as long as possible by sending **Keep-Alive** requests.

- The first list contains asynchronous tasks that consist in sending transactions to each listed node, continuously **if the node is configured in public mode only**:

This task runs in a loop as long as the target **Node** is accessible, the instance sends all the transactions in **MemPool** and keeps the progress so that they are not sent back to the target node.

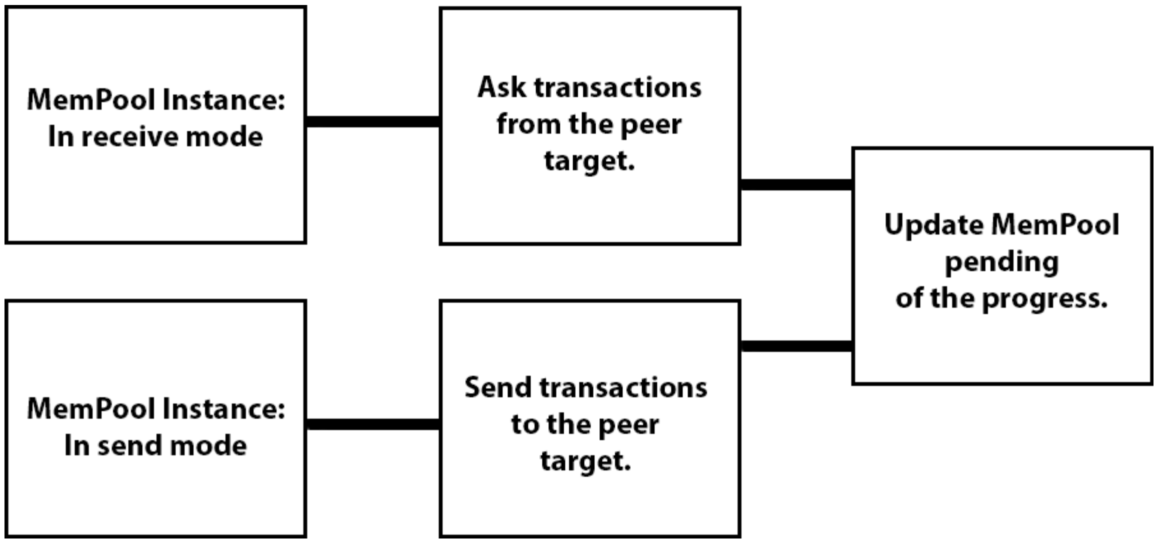
The instance gets a response from the target **Node** to take into account the reception of the sent transaction.

- The second list contains asynchronous tasks that consist of receiving transactions from each listed node in a continuous way:

These tasks consist of receiving a list of block heights contained in their MemPool from each of the targeted nodes, once their lists are received, if they contain a positive number of transactions, requests to receive the transactions are made from each of the instances.

The progress of each reception is saved so that they are not requested again later, at each new request, if a block height contains more transactions, the requests are restarted indicating the number already received, so that all the already synchronized transactions are not received again.

Here is a simple diagram of the two instances of broadcast MemPool targeting a public node:


MemPool Instance: In receive mode
Ask transactions from the peer target.
Update MemPool pending of the progress.
MemPool Instance: In send mode
Send transactions to the peer target.

## 1. Description of the voting system on the received contents:

When a synchronization item or a broadcast validation request are made,
The node performs a vote after checking the received content or validation response by listing the identical items in a list, the majority is taken into account, unless a number of nodes response is below a certain limit reconfigurable in the node settings file.

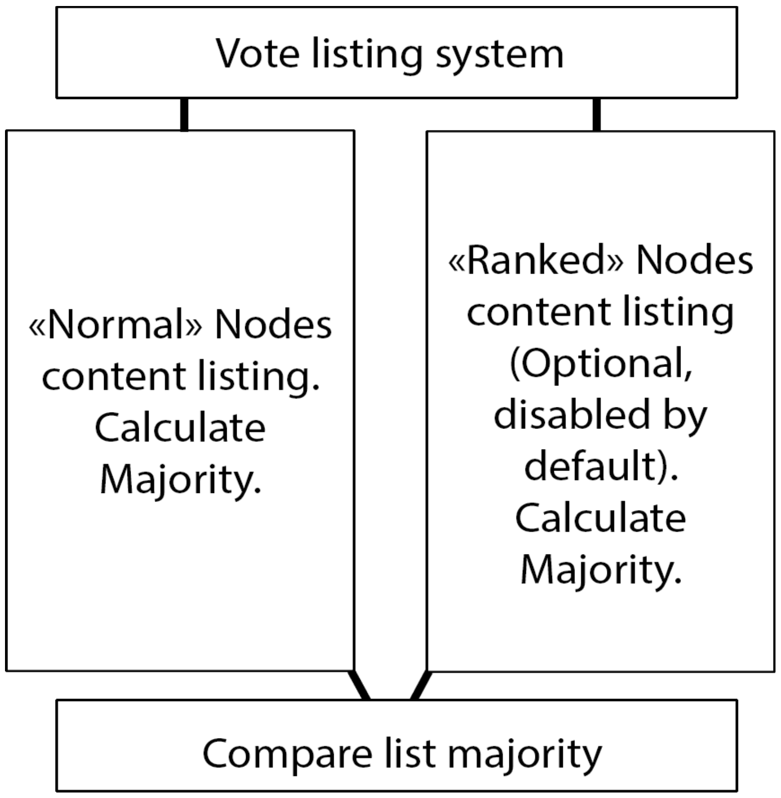
There is also another list, grouping the items received from nodes certified by one or more applied sovereign updates.
A verification of the signatures through the public keys referenced by these sovereign updates of the contents received from these nodes is performed.

Another thing, even if several nodes of the same IP send a response, only one of the nodes of the same IP will be taken into account in the vote.
It is the same for nodes certified by sovereign update, thus avoiding any cheating.

A comparison between the various lists of content is made, then a comparison between the lists of majority content between the "Normal" Nodes and the "Ranked" Nodes is made.

**The listing of certified nodes is optional, even if the sovereign updates are done automatically during the synchronization.**
**In case this option remains disabled, the listed nodes are considered as "normal" nodes.**

Here is a simple diagram describing the voting system:


Vote listing system
«Normal» Nodes content listing. Calculate Majority.
«Ranked» Nodes content listing (Optional, disabled by default). Calculate Majority.
Compare list majority

## II. Description of the internal transaction confirmation system.

A task dedicated to the confirmation of transactions is executed internally by the synchronization node,

### 1. Verification.

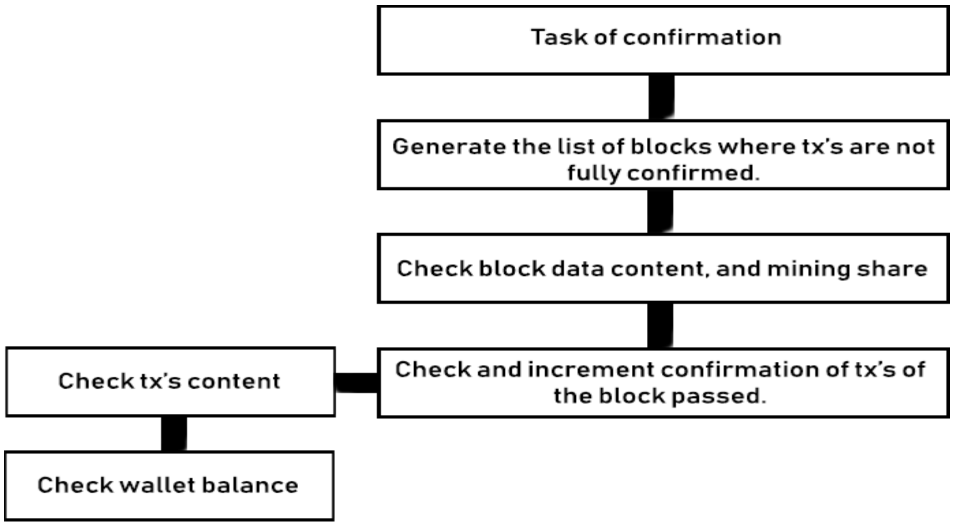
To begin with, a verification of the content, formats and values is performed.
A verification of the content and the signatures associated with this same content of a transaction is carried out, this verification occurs only once, i.e. at the time of the first confirmation.
A "light" representation contained in the transaction is regenerated to compare that the same representation provided is accurate, then a verification of the signature associated with this representation is tested.
Then a big hash, a representation is regenerated because not provided in the transaction to lighten its sending, storage. Only the signature of this representation is contained in the transaction, it is then tested with this representation.
This representation is composed of the elements of the transaction cut out every **1024 bytes**, to prevent the use of an exploit such as the **length attack**.

### 2. Wallet Balance.

At each necessary confirmation, a balance is calculated with respect to the previous transactions, to ensure that the amount sent is available until the moment of the last confirmation.

**Checkpoints** are generated to avoid starting from the first block to the last one, which speeds up the calculation of this balance.

The **checkpoints** are then stored in a cache containing the wallet addresses that have had at least one transaction, this cache is evolved using the hard disk so as not to use too much RAM.


Task of confirmation
Generate the list of blocks where tx's are not fully confirmed.
Check block data content, and mining share
Check and increment confirmation of tx's of the block passed.
Check tx's content
Check wallet balance

## III. Description of the sovereign update system.

The sovereign update system uses the private developer key to sign sovereign updates.
They allow to make changes on the Blockchain in a controlled way, without modifying the content, nor the transactions.
However, they allow to make the following changes:

- Assign a "SEED" rank on a synchronization node, allowing it to have more power on synchronization votes, it is only taken into account if a node in question activates the next mode, this "power" is simply ignored.

- Remove the "SEED" rank on a synchronization node.

- Change the signature of the developer assigned by default in the configuration of the BlockchainSetting.cs class without editing the code.

- To be able to make changes to the mining algorithm configuration.
To guarantee the maximum propagation of a sovereign update, a block height is requested, it allows to activate from the programmed height.

An entire system of monitoring sovereign updates ensures the progress of all the changes made, since a single change in the mining configuration changes all the necessary and future results.

The same goes for the change of the developer's signature.

## I. Constraints & potential problems.

Unlike other crypto-currencies that have not chosen to expose the potential constraints and problems of decentralization, these are exposed here.

There are various ways to compromise a decentralized system, any decentralized cryptocurrency can deal with them, it all depends on the means available to implement them.

The problems and constraints exposed do not allow to edit, submit invalid transactions, nor to be able to modify the balances of the wallets, since unlike other decentralized cryptocurrencies, the verifications are done internally in a parallel task after unlocking the blocks, an attack of 51% of the total hashrate, will therefore not allow "double spending".

However, if a crypto-currency faces this, it can have quite large consequences on its operation, to be honest, it was better to describe them to avoid unpleasant surprises.

**The system of sovereign update, allowing to certify nodes, can limit or even block this kind of attack, without affecting the decentralization, since as "normal" nodes, are also affected like these with the same rules of operation.**
**Given the skepticism of users, the separate voting system for certified nodes is disabled by default.**

### 1. Compromise the broadcast system.

If the network is not large enough, i.e., with a small number of publicly available nodes that interact with each other, or if an ill-intentioned person has the financial and technical means to compromise the broadcast system, this could affect it slightly or even greatly.

Let's say that the network in question consists of only 5 public nodes, owned by 5 different individuals, if a malicious person would like to influence the broadcast, he could obviously execute a larger number of public nodes and ignore any broadcast transactions.

Then to finalize this attack, it would be enough to use a hashrate more important than the average to compromise the broadcast of the pending transactions and thus to "cancel" them from being taken into account on their target height blocks.

Nevertheless, the attack in question is not infallible, an attempt can result from a virtual fork (copy) and the nodes in question held by the malicious person would be outside the network.

This attack can also compromise the broadcast of mining shares, if the majority held by this person ignores them, and only takes into account his own, this could influence the priority of progress of the blockchain, obviously, it must be done quickly enough with a consistent hashrate to achieve this.

### 2. Compromise the synchronization system.

In the same case explained above, a network that is not big enough could be the target of this kind of attack.
If a malicious person wants to compromise the synchronization of newcomers, i.e. from a point lower than the one currently available on the network, it is possible to have another blockchain synchronized, since it is the new majority available.

This only works if the user or the node being synchronized is at a much lower synchronization point than the current one.

## II. Presentation of the available tools.

Here is the list of available tools, they are developed using the reference **Xenophyte-Lib.dll**.

This reference contains a set of classes, allowing to generate the necessary database, to execute the memory management, to execute an instance of node synchronization internally and many other classes.

**Note that the presented tools will evolve, that their appearances may change during development, updates in the future.**
**All the tools have the name Xenophyte, as it will be updated by all these tools presented.**

### XENOPHYTE - Desktop Wallet :

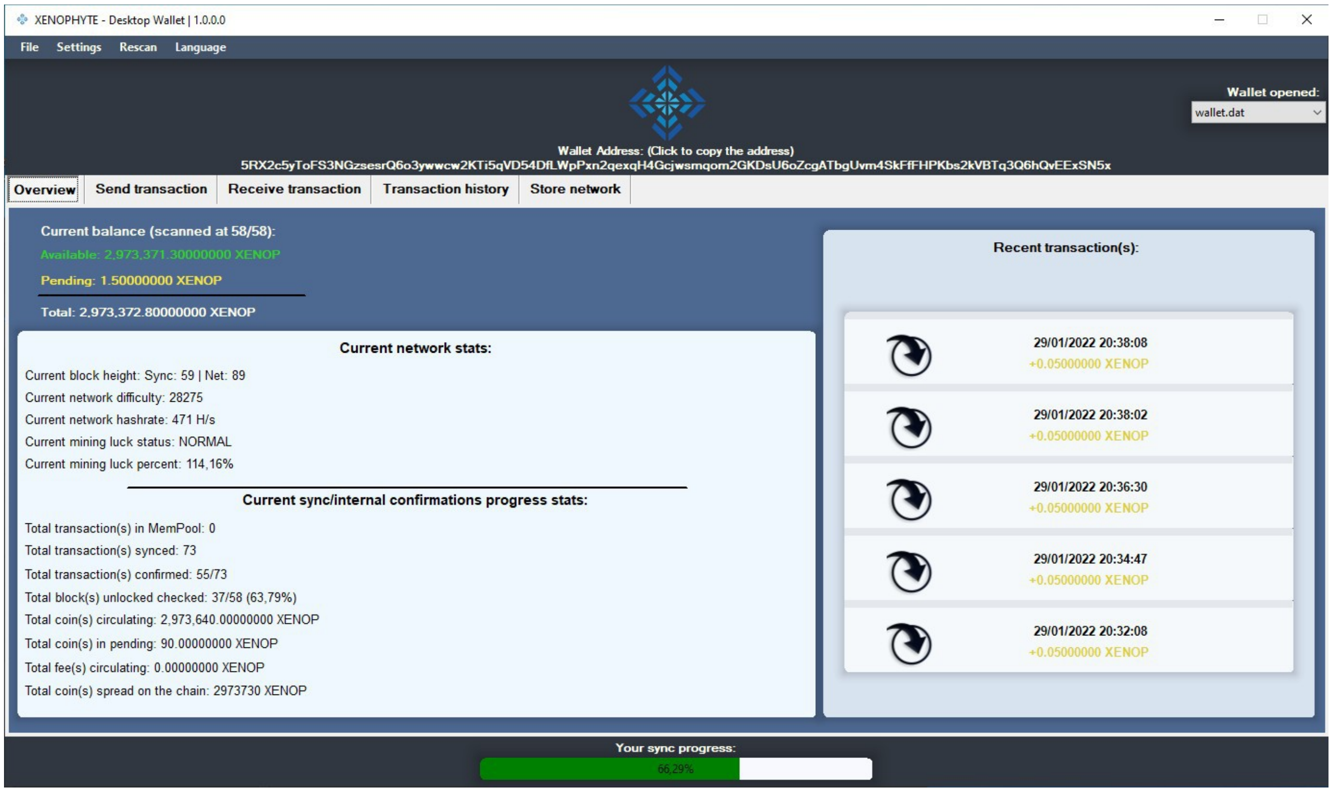
This is the default Desktop Wallet provided to users, it runs an instance of a synchronization node, configured as being closed from the outside, without participation in the network.

It is possible to configure it to make its synchronization instance accessible to the public to participate in the network, if the user's bandwidth allows it of course.

It is also possible to configure the Desktop Wallet to connect to an external or locally hosted Node, however it is recommended to always use your own synchronization Node.

The design has been programmed with custom **Winform** controls, some graphical effects like rounded edges on panels, displayed bitmaps, as well as shadow effects have been programmed and displayed with the default **Winform** graphics engine, **GDI+**.

The displayed transaction histories are also virtual instances created completely with Rectangles, Bitmaps, Graphics and math calculations.
The instances of the transaction histories are contained in memory and updated in the background.



Wallet opened: wallet.dat

Wallet Address: (Click to copy the address)
5RX2c5yToFS3NGzsesrQ6o3ywwcw2KTi5qVD54DfLWpPxn2qexqH4GcjwsmqOm2GKDsU6oZcgATbgUvm4SkFfFHPKbs2kVBTq3Q6hQvEExSN5x

Current balance (scanned at 58/58):
Available: 2,973,371.30000000 XENOP
Pending: 1.50000000 XENOP
Total: 2,973,372.80000000 XENOP

Current network stats:
Current block height: Sync: 59 | Net: 89
Current network difficulty: 28275
Current network hashrate: 471 H/s
Current mining luck status: NORMAL
Current mining luck percent: 114,16%

Current sync/internal confirmations progress stats:
Total transaction(s) in MemPool: 0
Total transaction(s) synced: 73
Total transaction(s) confirmed: 55/73
Total block(s) unlocked checked: 37/58 (63,79%)
Total coin(s) circulating: 2,973,640.00000000 XENOP
Total coin(s) in pending: 90.00000000 XENOP
Total fee(s) circulating: 0.00000000 XENOP
Total coin(s) spread on the chain: 2973730 XENOP

Recent transaction(s):
29/01/2022 20:38:08 +0.05000000 XENOP
29/01/2022 20:38:02 +0.05000000 XENOP
29/01/2022 20:36:30 +0.05000000 XENOP
29/01/2022 20:34:47 +0.05000000 XENOP
29/01/2022 20:32:08 +0.05000000 XENOP

Your sync progress:
66,29%

## XENOPHYTE - Peer:

This is a tool running the synchronization node instance, it allows to synchronize the blockchain, do solo mining, link external tools such as Desktop/RPC Wallet, participate in the network by configuring it as a public node.
It is the essential tool to make the decentralized network work.

## XENOPHYTE - Solo Miner :

The Solo Miner, allows you to participate in the network by performing the mining process to unlock blocks, each block successfully unlocked gives a certain number of cryptocurrency, depending on the established configuration and the ongoing progress of the Blockchain.
This tool is essential to progress the Blockchain, taking care of the pending transactions, the more different miners participate, the more the amount of estimated hashrate increases, the stronger the network will be against potential attacks stated in **chapter VI**.

## XENOPHYTE – RPC Wallet:

The RPC Wallet allow the user to handle multiple wallets, to send transaction by an API and can be supported by exchange.