nanoblocks

Release 0.0.1-alpha

Iván de Paz Centeno

Jan 12, 2022

CONTENTS

nanoblocks is an object-oriented API implementation of the Nano RPC that eases the communication with the Nano protocol.

With *nanoblocks* it is possible to integrate Nano cryptocurrency in any software with just a few lines of code, almost out-of-the-box. Check out the following documents to know how to start:

CHAPTER

ONE

GETTING STARTED

1.1 What is Nano?

Nano is a **decentralized cryptocurrency** that was born in 2015 under the wings of the Nano Foundation, led by Colin LeMahieu. It is claimed to be one of the fastests cryptocurrencies in the cryptomarket, as **the transactions can be instant and fee-less**. Furthermore, **there is no mining required**, all the coins are already distributed and it can be considered one of the **greenest cryptocurrencies available**.

There are a total of 133.248.290 Nano coins available, even though each can be divided up to 30 decimals. This division makes it suitable for microtransactions.

1.1.1 How does Nano work?

Nano is formed by a set of servers (called nodes) running the Nano node software. It is open-sourced (under the License BSD 3-Clause), meaning that every person can contribute to the code and host a node server.

All the nodes are interconnected forming the *Nano network*. Every node in the network contains the full ledger of transactions done since the very first transaction. Every time a transaction is attempted, all the nodes negotiate among them the validity of the transaction until the majority (66%) agree in confirmation, which is also known as Open-Representative-Voting. For this reason, Nano is considered a Peer2Peer decentralized cryptocurrency.

Unlike other cryptocurrencies where there is a single blockchain for all the operations, in Nano there are as many blockchains as accounts in the network. Furthermore, all the blockchains are ruled by a single structure based on a Direct Acyclic Graph (DAG)

> which is called **block-lattice**.

Nano is scalable due to every account ruling its own blockchain and no other's. However, every block that is going to be inserted in any blockchain must be confirmed by most of the nodes at any time. This makes the network secure and avoids double spend problems.

1.1.2 How to interact with the network?

The entry point to the network is the Nano node. In order to check accounts and make transactions, the Nano node software usually publish an API in the http (TCP 7076) and websocket (TCP 7078) protocols. The node operators usually restrict the access to these node APIs to avoid saturation and hacking. However, there is still a certain set of public API exposed through SSL layers, like for example the ones exposed in https://publicnodes.somenano.com/.

Every wallet software, like Natrium, Nalli or Nault, use a private (sometimes public) node to operate. For this reason, if the wallet's node shuts down, the wallet might lose the service. However, since the account data is stored in every node's ledger, its control can be retrieved by using any other wallet software, or even your own wallet built with nanoblocks.

1.1.3 Can I host a node?

Definitely Yes, and you should if you want to have a serious development with this library, even though it is not mandatory. For experimental setups, public nodes can be used instead; however, it is highly encouraged to host your own node due to security concerns. Also, some operations can be applied offline where no nodes are required at all.

nanoblocks points to a public node by default (Mynano.ninja), which allows to be used out-of-the-box. In case more advanced configuration is required, a tutorial on how to set up a node can be found here.

1.2 Getting started

In order to start, make sure a communication to a node RPC http and WebSocket is available. Otherwise, most of the examples presented in this page won't work properly.

Check the following section to know how to install a node.

1.2.1 Set-up a Node

nanoblocks can release all its functionality by accessing a Node API. You can either get access to a public Node http REST API and WebSockets, or mount your own node instead (which is highly encouraged). If you want to install the Nano node, you should try to set up a Node by following the official installation guide.

Take note of the rest API address *(usually TCP node_host:7076)* and the WebSocket address *(usually TCP node_host:7078)*, as you are going to need them from now on.

Also, note that having a dedicated work server for generating work hashes is highly encouraged. check out the nano work server. Each Nano transaction requires a work hash attached, and generating the hash is a heavy process which usually requires a dedicated GPU to fit in time. Even though this problem can be circumvented by caching the work hash for each account frontier, having a dedicated work server separated from the node is still encouraged.

1.2.2 Interacting with the network

The entry point to the network is the class `NanoNetwork`:

```
>>> from nanoblocks.node import NanoNode
>>> from nanoblocks.network import NanoNetwork

>>> node = NanoNode(rest_api_url="http://localhost:7076", websocket_api_url="ws://
↪localhost:7078")
>>> node
[Node http://localhost:7076 (Nano V21.3)]

>>> network = NanoNetwork(node)
>>> network
[Node http://localhost:7076 (Nano V21.3)] (270 peers; 15362838 account)
```

The *network* object contains access to three basic members: the *blocks* in the network, the *accounts* globally registered and the *wallets*.

1.2.3 Accessing an account

One of the most interesting functionalities in most criptocurrencies is that the ledger is public and all the accounts can be accessed. Accessing an account means reading its balance and its blockchain (history of transactions).

Note that *accessing an account* != *controlling an account*, as for controlling an account it requires you to have the corresponding *private key* (which is different from the wallet seed!).

Given the *network* object, accessing an account can be done as follows:

```
>>> account = network.accounts["nano_
↪39a73oy5ungrhxy5z5oao1xso4zo7dmgpjd4u74xcrx3r1w6rtazuouw6qfi"]
>>> account
nano_39a73oy5ungrhxy5z5oao1xso4zo7dmgpjd4u74xcrx3r1w6rtazuouw6qfi (
    Total blocks: 733
    Total balance: 0.000002000000000000000000000002 NANO
    Confirmed balance: 0.000000000000000000000000000000 NANO
    Pending balance: 0.000002000000000000000000000002 NANO
    Last confirmed payment: 2020-12-02 01:30:39+01:00
    Is virtual: False
)
```

You might have noticed well: *network.accounts[]* behaves like a python dictionary, which in turns makes it easy to access any account in the network. In *nanoblocks*, every account is wrapped inside the class `Account` which gives basic functionality for account handling:

```
>>> account.balance
0.000002000000000000000000000002 NANO

>>> account.pending_balance
0.000002000000000000000000000002 NANO

>>> account.confirmed_balance
0.000000000000000000000000000000 NANO

>>> account.public_key
9D050D7C3DD1D87F7C3F8EA8A83B9A8BF52AE6EB4562D945D563A1C0384C691F

>>> account.frontier
FED268136D2931EDEA61057D91C3C250894EF95C14C0D16CA0D126D99579C53C

>>> account.representative  # The representative is another account object
nano_16u1uufyoig8777y6r8iqjtrw8sg8maqrm36zzcm95jmbd9i9aj5i8abr8u5 (
    Total blocks: 6
    Total balance: 0.000000000000000000000000000000 NANO
    Confirmed balance: 0.000000000000000000000000000000 NANO
    Pending balance: 0.000000000000000000000000000000 NANO
    Last confirmed payment: 2020-12-02 01:57:11+01:00
    Is virtual: False
)
```

1.2.4 Accessing a block

If you know a block hash and you want to check its information, it can be done in a similar way than with accounts, but with the *blocks* member:

```
>>> block = network.blocks[
↪"4FEC4BDD078C741F599221C67C8BE6493C872EF9B30968BBF4991640FFF42DA2"]
>>> block
[Block #4 from account nano_39a73oy5ungrhxy5z5oao1xso4zo7dmgpjd4u74xcrx3r1w6rtazuouw6qfi]
    Type: receive
    Hash: 4FEC4BDD078C741F599221C67C8BE6493C872EF9B30968BBF4991640FFF42DA2
    Source account: nano_39a73oy5ungrhxy5z5oao1xso4zo7dmgpjd4u74xcrx3r1w6rtazuouw6qfi
    Amount: 0.000040000000000000000000000000
    Local date: 1970-01-01 01:00:00+01:00
```

Every Block is wrapped inside the class derived from `Block`, which can in turn be a `BlockSend`, a `BlockReceive` or a `BlockState`. The main difference between each *block* implementation is the arrangement of the fields and the way they are displayed when printed.

Check the block docs to know what methods and attributes are available for each.

1.2.5 Accessing a wallet

Wallets can be accessed by using the property *wallets* of the *network* object, which gives access by the seed or the bip39-mnemonic 24-word phrase.

```
# To access an existing wallet by using the 64-Bytes seed:
>>> wallet = network.wallets[
↪"7F632A80ECCC54A058602CD64A81D23A6B4D7320562E4767C9EB0BBB1151CDF2"]

# Alternatively, it can be accessed with the BIP-39 24 words:
>>> wallet = network.wallets[['legal', 'bone', 'parent', 'sunset', 'shed', 'expand',
↪'ghost', 'airport', 'stone', 'favorite', 'innocent', 'inquiry', 'regular', 'ridge',
↪'life', 'shift', 'electric', 'dinner', 'kiss', 'blast', 'rain', 'pottery', 'daughter',
↪'execute']]

# Wallet information can be printed out
>>> print(wallet.seed)
7F632A80ECCC54A058602CD64A81D23A6B4D7320562E4767C9EB0BBB1151CDF2

>>> print(wallet.mnemonic)
['legal', 'bone', 'parent', 'sunset', 'shed', 'expand', 'ghost', 'airport', 'stone',
↪'favorite', 'innocent', 'inquiry', 'regular', 'ridge', 'life', 'shift', 'electric',
↪'dinner', 'kiss', 'blast', 'rain', 'pottery', 'daughter', 'execute']
```

Creating new wallets

New wallets can be created with a single line of code:

```
>>> new_wallet = network.wallets.create()
```

You can then print the seed and/or the mnemonic BIP39 24-word list from it.

The creation of the wallet relies on the random seed number generator from the operating system, which is considered to be cryptographically secure.

Note that this method does not require to have a node attached. This means that it can run completely offline (even without internet):

```
>>> from nanoblocks.network import NanoNetwork
>>> network = NanoNetwork()  # No node attached
>>> wallet = network.wallets.create()
```

This happens because *nanoblocks* integrates all the cryptographic functions required to create wallets and accounts.

Creating wallet accounts

Wallets are the basic building block in Nano, as they allow you to create accounts. Every wallet can create 2^32 accounts, which is an extremely big number (4294967296). Every account in a wallet is deterministically indexed by an integer in the range [0, 2^32]. They can be easily created as follows:

```
>>> account_0 = wallet.accounts[0]
>>> account_1 = wallet.accounts[1]
```

The account at every index is always the same account, no matter which software wallet you use. This means that the wallet at a given index is the same in nanoblocks, in Natrium, Nault or any other wallet software. Note that this process can still be done offline, as it is not required nodes to create accounts.

When an account is new and doesn't have blockchain, it is considered *virtual*. A virtual account becomes real in the ledger of the nodes as soon as it publish a *BlockReceive*, which requires someone sending to it a *BlockSend* with some amount first.

All the accounts accessed through a wallet are automatically unlocked with the corresponding *private key*. This allows you to create and sign blocks in its blockchain. You can check the private key of an account as follows:

```
>>> account_0_privkey = account_0.private_key
```

Furthermore, if you have the private key, you can unlock it at any time directly without the need of the wallet:

```
>>> account_0 = network.accounts['account_0_address']
>>> account_0.unlock(account_0_privkey)
```

1.2.6 Requesting payments

With *nanoblocks*, requesting a payment for an account can be simplified in two lines of code. Any *Account* in the network can be used to request a payment through its method *request_payment()*. When invoked, a Nano amount is passed as parameter and a `Payment` object is returned, which gives an easy interface to the payment process.

```
>>> account = network.accounts['nano_
↪1nween66fcspgkx33defgtmypgzkqf4heihaubqwjyhjrwma5qz4z9r45szj']
>>> payment = account.request_payment("0.1") # In Nano units
```

The *Payment* object can be used to generate a payment link

```
>>> print(payment.uri)
nano:nano_1nween66fcspgkx33defgtmypgzkqf4heihaubqwjyhjrwma5qz4z9r45szj?
↪amount=100000000000000000000000000000000
```

It can also generate QR codes as PIL images, which can be scanned by any modern wallet software like Natrium or Nault:

```
>>> qr_image = payment.qr_code
>>> qr_code.show()
```

(ps: if you like the project, donations are accepted by using this very same QR code image!)

And the most interesting method of the *payment* object is the *wait()* method, which allows to lock the thread until a payment is detected (or until the timeout raises):

```
>>> block = payment.wait(timeout=30)
>>> block
```

The *wait()* method accepts a *timeout* parameter in seconds. When the payment is processed, the corresponding *SendBlock* is immediately returned back to you. This block is useful as you can track the confirmation of the block and build the corresponding *ReceiveBlock* to convert the pending transaction in confirmed transaction, in case you have control over the account.

Note that no private keys are involved in the process yet, meaning that **any** account can be used for this operation.

1.2.7 Sending and receiving Nano

Sending and receiving Nano are tightly coupled with block handling and work generation. Since it is a little more complex (not so much!) than the concepts and uses explained here, it has its own document. Everything is explained in the following section, so take a breath first before diving!