Exchange Integration Guide

Contact mica.busch@raiblocks.net with any questions regarding integration on your exchange

Introduction

RaiBlocks operates under a completely new model in comparison with the Satoshi wallet RPC design.

- Each private key corresponds to only one public address.
- Every address has its own block-chain, the equivalent of its transaction history.
- Each node contains a complete copy of the global ledger, automatically providing indexing of accounts.
- Transactions are published to the network immediately and are immutable once confirmed by peers.

We do recommend installation of the node as a service on your server, as maintenance is made much easier. Initial sync of the global ledger can be slow as the process is currently disk-IO bound. It is recommended to use high-speed SSDs or hardware RAID with plenty of cache. For infrastructure that provides more than 6GB of RAM, it is possible to create a ram-disk in memory and creating a file system link from the data directory to the ram-disk. This has been shown to significantly speed up sync, with a full sync from fresh installation taking as little as 30 minutes. Precautions must be taken however if this is used in a production environment, that persistence to disk is taken care of. Once synchronized with the rest of the network, it is advised that the data folder be kept on a hard disk or SSD to best protect the data, also because once the node is synced disk-IO is no longer a factor in performance.

A node installation can be used either to fully manage accounts within itself as a standard wallet, or used as a bridge to the network for a third-party wallet. Third party software may use the node as a block explorer to fetch data on accounts, get block details, and gather all needed information to create blocks. Once created, blocks can then be submitted to be verified, broadcast to the network, and confirmed using the node.

Integration

RPC Communication is handled through simple TCP requests in JSON format. Replies are also delivered in JSON format. Command-line requests can be done using curl or other such tools. Example:

```
curl -g -d '{ "action":"key_create"}' [::1]:7076
```

Here we use curl to send a JSON formatted request to the loopback address. Port 7076 is reserved for RPC commands. In the node's config.json file both "rpc_enable" and "enable_control" must be set to "true" for wallet control and accounting. If using the node only as a RaiBlocks network interface, "enable_control" can be "false". The destination address must match the interface address given in the configuration file. For basic testing, an interface address "::1" for localhost will work fine. If node is on a separate server, you must bind to the network interface address other clients will be connecting to. If the command is accepted, either data or a reply of success will be returned in JSON format, or an error message is given if the command is invalid.

All requests are made via HTTP client POST to port 7076, with the request body comprising only the JSON formatted action. Replies are similarly processed in return, with the data or status given in the reply's body.A callback feature provides a way to process incoming blocks without polling. In the wallet's config.json file the following parameters may be set to provide this event-driven feedback:

```
"callback_address": "127.0.0.1", "callback_port": "17076", "callback_target": "/"
```

The callback sends a POST in JSON format to the address:port and target given in the configuration. Format is as follows:

```
{       'amount': '<transaction amount in raw units>',
        'account': <account address block belongs to>',
        'block': {      'previous': '<previous block hash in account chain>',
                        'work': '<proof of work>',
                        'type': '<open, change, send or receive>',
                        'source': '<block hash of send block>',
                        'signature': '<valid signature to verify against>' },
        'hash': '<this blocks hash>'    }
```

Units of Value

RaiBlocks stores balances as a 128-bit unsigned integer value. The maximum number of units of value is therefore:
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF (hexadecimal)
340,282,366,920,938,463,463,374,607,431,768,211,455 (decimal)

It is apparent this is far too cumbersome for users to work with. Thus, we use SI units to make reference to the values stored in an account.

Gxrb = 1000000000000000000000000000000000 (10^{33}) raw units
Mxrb = 1000000000000000000000000000000 (10^{30}) raw units
kxrb = 1000000000000000000000000000 (10^{27}) raw units
xrb = 1000000000000000000000000 (10^{24}) raw units
mxrb = 1000000000000000000000 (10^{21}) raw units
uxrb = 1000000000000000000 (10^{18}) raw units

At this point in time, it is adopted by users and exchanges that 1 XRB is equal to 1 (**M**illions)xrb, or 1 * 10^{30} raw units. The wallet makes use of a selectable divider for ease of use – at some point it may be necessary to move from Mxrb to kxrb, and adjustments will be made in the wallets and on exchanges, but the ISO exchange ticker will remain XRB, regardless of divider.

Address & Account Creation

Accounts held within the wallet can be added by deterministic generation (seed), or randomly generated (ad-hoc). Nodes may transmit a transaction to addresses that have not yet been opened on the global ledger, as long as the address is formatted correctly. Internally, an address is represented as a 256-bit unsigned integer – there are no invalid addresses in this format, any value is valid. An address is expanded into a 260-bit number, padded with zeroes to the most significant digit. Then the address is encoded using visually easy to differentiate letters in 5-bit segments. The unencoded address is also hashed once using Blake2B with an 8-bit digest. The resulting hash is also encoded.

Raw Address	8933B4083FE0E42A97FF0B7E16B9B2CEF93D31318700B328D6CF6CE931BBF8D4 256-bit unsigned integer in hexadecimal format
Encoded Address	xrb_34bmpi65zr967cdzy4uy4twu7mqs9nrm53r1penffmuex6ruqy8nxp7ms1h1 Encoded version of raw address as a 260-bit integer, each character represents 5-bits of the address value. As the beginning of the number is zero-padded, only ‘1’ or ‘3’ are valid first characters.

Prefix	Address	Checksum
xrb_	34bmpi65zr967cdzy4uy4twu7mqs9nrm53r1penffmuex6ruqy8n 260 bits	xp7ms1h1 40 bits

Binary	Encoded	Binary	Encoded	Binary	Encoded	Binary	Encoded
00000	‘1’	01000	‘a’	10000	‘i’	11000	‘r’
00001	‘3’	01001	‘b’	10001	‘j’	11001	‘s’
00010	‘4’	01010	‘c’	10010	‘k’	11010	‘t’
00011	‘5’	01011	‘d’	10011	‘m’	11011	‘u’
00100	‘6’	01100	‘e’	10100	‘n’	11100	‘w’
00101	‘7’	01101	‘f’	10101	‘o’	11101	‘x’
00110	‘a’	01110	‘g’	10110	‘p’	11110	‘y’
00111	‘b’	01111	‘h’	10111	‘q’	11111	‘z’

An account establishes its chain with an open block, which is the equivalent to a change and receive block all in one. It both receives the first transaction and lets the network know which representative this account’s balance is delegated to.

To create an account using the node, you must first create a wallet:

```
curl -g -d ‘{ "action": "wallet_create" }’ [::1]:7076
```

A new wallet ID is returned, you will need to reference this wallet whenever working with accounts within the node. The node may contain multiple wallets, each with separate accounts within. Every wallet has its own randomized seed which new accounts are deterministically generated from. Wallets may also contain imported or randomly generated accounts not related to the wallet seed. Each wallet may also be password encrypted.

Change wallet password:

```
curl -g -d '{ "action": "password_change", "wallet": "<wallet id>", "password": "<password>" }' [::1]:7076
```

Enter wallet password to unlock:

```
curl -g -d { "action": "password_enter", "wallet": "<wallet id>", "password": "<password>" }' [::1]:7076
```

Check if password is valid:

```
curl -g -d '{ "action": "password_valid", "wallet": "<wallet id>" }' [::1]:7076
```

Once a valid wallet is configured, individual addresses may be created. Please note, RaiBlocks does not have the capability of creating multiple addresses for the same account or private key. Each address is paired to a unique private key.

Transactions

Every completed transaction consists of two blocks. RaiBlocks does away with the idea of confirmations as well as pending transactions in a mempool waiting to be confirmed in the next mined block. No such mempool exists with RaiBlocks as every transaction, once validated by the local node, is transmitted to all connected peers for confirmation. Nodes then immediately reply with a confirmation or request to vote if a potential fork is detected.

If two blocks are received that reference the same previous or source block, a potential fork has been detected. A fork vote is broadcast to the network and all online representatives respond with a vote equivalent to their balance or the sum of the balance of their delegates, as well as the hash of the block they saw first. The correct block will have the majority of the votes from nodes and will be retained in the global ledger. The block that loses the vote is discarded.

NOTE: *The callback feature of the node will relay all blocks as they are received. Forks are resolved only after a vote, so if processing blocks from the callback stream make sure blocks with identical source or previous block referenced are tested against the node's ledger.*

In some circumstances, brief network connectivity issues may cause a broadcasted block to not be accepted by all peers. Any block that succeeds this block *on this account* will be ignored as invalid by peers whom did not see the initial broadcast. To remedy this, a simple rebroadcast can be done of the block to the network, and the network will retrieve the succeeding blocks automatically. Even when a fork or missing block occurs, only the accounts referenced in the transaction are affected – the rest of the network proceeds with processing transactions for all other accounts.

Address Balance

Account balance is recorded within the ledger itself. Rather than record the amount of the transaction in a single block, verification requires checking balance at the send block and previous, calculating the difference. Then the receiving account may increment the previous balance as measured into the final balance given in the new receive block. This is done to improve processing speed when downloading high volumes of blocks. When requesting account history, amounts are given already calculated for you. When processing blocks manually, the calculating procedure must be done yourself.

To find the balances relating to an address:

```
curl -g -d '{ "action": "account_balance", "account": "<address>" }' [::1]:7076
```

Please note, this will return both a balance and pending balance in decimal. This also works on any address in the entire network, not just local accounts.

Sending from an Address

To send from an address, the address must already have an existing open block, and therefore a balance. A send block is immutable one confirmed and cannot be canceled. Once broadcast to the network, funds are deducted from the balance of the sender's account and wait to be accepted by the receiving party. Funds cannot be recovered except by a corresponding receive block signed by the recipient account, and then sent back again.

To create a send block and thus start a transaction:

```
curl -g -d '{ "action": "send", "wallet": "<wallet id>", "source": "<from address>", "destination": "<to
address>", "amount": "<amount in decimal raw>" }' [::1]:7076
```

The reply to this command, if successful will be the block hash of the newly created block. Blocks may also be created manually for any account (as long as it is signed by the proper private key) and transmitted from any node, even if that account or address is not held by that node. The same verification of work and signature takes place as with a local wallet account, and the block is then broadcast to the rest of the network. In this way you can handle

accounts outside of the node, using your own software. Sending of transactions is rate-limited by a simple proof of work requirement. This was put in place to limit harmful transaction spam (since there are no transaction fees) but requires additional examination in an exchange environment – see the section "[Proof of] Work" for more details.

Receiving for an Address

To complete a transaction, the recipient of sent funds must create a receive block on their own account-chain. Once this block is created and broadcast, their account's balance is updated and the funds have officially moved into their account. Please note, too, that "pending" or "not pocketed" funds should not be considered awaiting confirmation. They are as good as spent from the sender's account, and the sender cannot revoke that transaction. Thus, crediting a user's account can be done as soon as the send block is detected and no fork votes immediately follow. The exchange service may take the time to create the receive block (requiring proof of work) whenever load is low and then send the funds to the hotwallet (another send & receive pair, both also requiring proof of work to be generated).

If the wallet containing the pending blocks is unlocked, pending transactions will be automatically received as proof of work is made available and the transaction is higher than the "receive_minimum" value as set in config.json.

To check for pending transactions (incomplete send blocks pointing to our address):

```
curl -g -d '{ "action": "pending", "account": "<address>", "count": "<how many results to return>" }' [::1]:7076
```

If there are any pending transactions, a list of block hashes will be returned. To create a receive block and complete a transaction:

```
curl -g -d '{ "action": "receive", "wallet": "<wallet id>", "account": "<receiving address>", "block": "<hash of send block>" }' [::1]:7076
```

As in the case of the send command, once successful the command returns the hash of the new block.

Representatives and Forks

Consensus and fork protection is provided by representative nodes. Any address can be a representative – what makes it effective is when the node containing that account address is kept online 24/7 and allowed to vote. The voting weight of a representative's address is given to it by the accounts that name it as their representative. As an exchange, it is usually your hotwallet that is also your representative address. Any accounts created under your node should have this address selected as their representative, and thus your node will be able to help vote on forks with the full weight of the funds your users entrust to you.

How to check what an account's representative is:

```
curl -g -d '{ "action": "account_representative", "account": "<address>" }' [::1]:7076
```

How to check the voting weight of an account address:

```
curl -g -d '{ "action": "account_weight", "account": "<address>" }' [::1]:7076
```

How to change the representative of an account in your wallet:

```
curl -g -d '{ "action": "account_representative_set", "wallet": "<wallet id>", "account": "<address>", "representative" : "<representative address>" }' [::1]:7076
```

If you wish to set the default representative for all new accounts within a wallet:

```
curl -g -d '{ "action": "wallet_representative_set", "wallet": "<wallet id>", "representative": "<address of new representative>" }' [::1]:7076
```

[Proof of] Work

Networks that use serial mining of blocks require a consistent average of time elapsed between blocks. To accomplish this, these networks use a proof of work system with distributed methods of adjusting the calculation difficulty in order to facilitate consistent block-times relative to total computational power being applied to mining. This method is both time consuming as well as wasteful of energy

With the creation of RaiBlocks, serial block-chains are no longer required for the entire ledger of thousands of accounts to maintain consensus. Instant transactions require a different model of function, and with instant transactions comes the threat of transaction spam. To prevent misuse of the network's resources, each transaction submitted and broadcast must provide a valid proof of work. This proof is of constant difficulty, and is based on the block hash of the last block in the block generator's account. In this manner, every account is limited from repeatedly and without cost transmitting miniscule transactions to the network to process. Instead of using a transaction fee, RaiBlocks requires this cost be paid in computational resources. Using this serial generation method allows the pre-caching of valid proof of work for the next transaction, allowing instant single transactions with minimal delay for all following transactions on the same account-chain. Requiring both the sending and receiving parties to produce valid proof of work for their respective send and receive blocks also distributes the computational load of a transaction network-wide so no one party bears the full burden.

How the proof of work is generated is out of the scope of this publication. However, the difficulty of the proof is sufficient to significantly slow down a computer hosting the node, should it be required to handle a high volume of transactions – such as an exchange does. To assist with this, the node does provide acceleration that can take advantage of OpenCL compatible GPUs. Below is a comparison of real-life benchmarks conducted:

Device	Transactions per second
Nvidia Tesla V100 (AWS)	6.4
Nvidia Tesla P100 (Google Cloud)	4.9
Nvidia Tesla K80 (Google Cloud)	1.64
AMD RX 470 OC	1.59
Intel Core i7 4790K AVX2	0.33
Intel Core i7 4790K WebAssembly (Firefox)	0.14
Google Cloud 4 vCores	0.14–0.16
ARM64 server 4 cores (Scaleway)	0.07

It is obvious that hardware acceleration is the route necessary to provide users with the faster transactions they want. While a user may be content waiting a few moments for their personal wallet to generate the required proof, the same user won't have similar patience when waiting for all the transactions an exchange must process before getting to *their* transaction. To this end, we do ask that exchanges review the prospect of upgrading their infrastructure with at least one GPU in a physical server or provision a GPU instance for high-load use. To assist exchanges with delivering high-speed transaction processing, we will also be renting use of our own dedicated infrastructure to process transaction proofs more quickly. If this is desired for your exchange until you can obtain such acceleration on a permanent basis, please contact us.

RPC Command Reference

All current RPC commands are listed at our Wiki: https://github.com/clemahieu/raiblocks/wiki/RPC-protocol

RPC Libraries are also available:

JavaScript: https://github.com/SergiySW/RaiBlocksJS

PHP: https://github.com/mikerow/easyraikitphp

Python: https://github.com/AuliaYF/easyraikit-python

Elixir: https://github.com/willHol/rai_ex