pyln-client Documentation

Christian Decker

Sep 19, 2020

Contents:

CHAPTER 1

API Documentation

class pyln.client.**LightningRpc**(*socket_path*, *executor=None*, *logger=<module 'logging' from '/home/docs/.pyenv/versions/3.7.3/lib/python3.7/logging/__init__.py'>*, *patch_json=True*)

RPC client for the *lightningd* daemon.

This RPC client connects to the *lightningd* daemon through a unix domain socket and passes calls through. Since some of the calls are blocking, the corresponding python methods include an *async* keyword argument. If *async* is set to true then the method returns a future immediately, instead of blocking indefinitely.

This implementation is thread safe in that it locks the socket between calls, but it does not (yet) support concurrent calls.

class LightningJSONDecoder(***, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*, *patch_json=True*)

static replace_amounts(*obj*)

Recursively replace _msat fields with appropriate values with Millisatoshi.

class LightningJSONEncoder(***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

default(*o*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
```

(continues on next page)

(continued from previous page)

```
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

autocleaninvoice (*cycle_seconds=None*, *expired_by=None*)
Sets up automatic cleaning of expired invoices. {cycle_seconds} sets the cleaning frequency in seconds (defaults to 3600) and {expired_by} sets the minimum time an invoice should have been expired for to be cleaned in seconds (defaults to 86400).

check (*command_to_check*, ***kwargs*)
Checks if a command is valid without running it.

checkmessage (*message*, *zbase*, *pubkey=None*)
Check if a message was signed (with a specific key). Use returned field ['verified'] to get result.

close (*peer_id*, **args*, ***kwargs*)
Close the channel with peer {id}, forcing a unilateral close after {unilateraltimeout} seconds if non-zero, and the to-local output will be sent to {destination}.

Deprecated usage has {force} and {timeout} args.

connect (*peer_id*, *host=None*, *port=None*)
Connect to {peer_id} at {host} and {port}.

decodepay (*bolt11*, *description=None*)
Decode {bolt11}, using {description} if necessary.

delexpiredinvoice (*maxexpirytime=None*)
Delete all invoices that have expired on or before the given {maxexpirytime}.

delinvoice (*label*, *status*)
Delete unpaid invoice {label} with {status}.

dev_crash ()
Crash lightningd by calling fatal().

dev_fail (*peer_id*)
Fail with peer {peer_id}.

dev_forget_channel (*peerid*, *force=False*)
Forget the channel with id=peerid.

dev_memdump ()
Show memory objects currently in use.

dev_memleak ()
Show unreferenced memory objects.

dev_pay (*bolt11*, *msatoshi=None*, *label=None*, *riskfactor=None*, *description=None*, *maxfeepercent=None*, *retry_for=None*, *maxdelay=None*, *exemptfee=None*, *use_shadow=True*)
A developer version of *pay*, with the possibility to deactivate shadow routing (used for testing).

dev_reenable_commit (*peer_id*)
Re-enable the commit timer on peer {id}.

dev_rescan_outputs ()
Synchronize the state of our funds with bitcoind.

dev_rhash (*secret*)
Show SHA256 of {secret}

dev_sign_last_tx(*peer_id*)
Sign and show the last commitment transaction with peer {id}.

dev_slowcmd(*msec=None*)
Torture test for slow commands, optional {msec}.

disconnect(*peer_id*, *force=False*)
Disconnect from peer with {peer_id}, optional {force} even if has active channel.

feerates(*style*, *urgent=None*, *normal=None*, *slow=None*)
Supply feerate estimates manually.

fundchannel(*node_id*, **args*, ***kwargs*)
Fund channel with {id} using {amount} satoshis with feerate of {feerate} (uses default feerate if unset). If {announce} is False, don't send channel announcements. Only select outputs with {minconf} confirmations. If {utxos} is specified (as a list of 'txid:vout' strings), fund a channel from these specifics utxos.

fundchannel_cancel(*node_id*)
Cancel a 'started' fundchannel with node {id}.

fundchannel_complete(*node_id*, *funding_txid*, *funding_txout*)
Complete channel establishment with {id}, using {funding_txid} at {funding_txout}.

fundchannel_start(*node_id*, **args*, ***kwargs*)
Start channel funding with {id} for {amount} satoshis with feerate of {feerate} (uses default feerate if unset). If {announce} is False, don't send channel announcements. Returns a Bech32 {funding_address} for an external wallet to create a funding transaction for. Requires a call to 'fundchannel_complete' to complete channel establishment with peer.

fundpsbt(*satoshi*, *feerate*, *startweight*, *minconf=None*, *reserve=True*, *locktime=None*)
Create a PSBT with inputs sufficient to give an output of satoshi.

getinfo()
Show information about this node.

getlog(*level=None*)
Show logs, with optional log {level} (info|unusual|debug|io).

getpeer(*peer_id*, *level=None*)
Show peer with {peer_id}, if {level} is set, include {log}s.

getroute(*node_id*, *msatoshi*, *riskfactor*, *cltv=9*, *fromid=None*, *fuzzpercent=None*, *exclude=[]*, *maxhops=20*)
Show route to {id} for {msatoshi}, using {riskfactor} and optional {cltv} (default 9). If specified search from {fromid} otherwise use this node as source. Randomize the route with up to {fuzzpercent} (0.0 -> 100.0, default 5.0). {exclude} is an optional array of scid/direction or node-id to exclude. Limit the number of hops in the route to {maxhops}.

getsharedsecret(*point*, ***kwargs*)
Compute the hash of the Elliptic Curve Diffie Hellman shared secret point from this node private key and an input {point}.

help(*command=None*)
Show available commands, or just {command} if supplied.

invoice(*msatoshi*, *label*, *description*, *expiry=None*, *fallbacks=None*, *preimage=None*, *exposeprivatechannels=None*)
Create an invoice for {msatoshi} with {label} and {description} with optional {expiry} seconds (default 1 week).

listchannels(*short_channel_id=None*, *source=None*)
Show all known channels, accept optional {short_channel_id} or {source}.

listconfigs(*config=None*)
List this node's config.

listforwards()
List all forwarded payments and their information.

listfunds()
Show funds available for opening channels.

listinvoices(*label=None*)
Show invoice {label} (or all, if no {label}).

listnodes(*node_id=None*)
Show all nodes in our local network view, filter on node {id} if provided.

listpayments(*bolt11=None*, *payment_hash=None*)
Show outgoing payments, regarding {bolt11} or {payment_hash} if set Can only specify one of {bolt11} or {payment_hash}.

listpeers(*peerid=None*, *level=None*)
Show current peers, if {level} is set, include {log}s".

listsendpays(*bolt11=None*, *payment_hash=None*)
Show all sendpays results, or only for *bolt11* or *payment_hash*.

listtransactions()
Show wallet history.

multifundchannel(*destinations*, *feerate=None*, *minconf=None*, *utxos=None*, *minchannels=None*, ***kwargs*)
Fund channels to an array of {destinations}, each entry of which is a dict of node {id} and {amount} to fund, and optionally whether to {announce} and how much {push_msat} to give outright to the node. You may optionally specify {feerate}, {minconf} depth, and the {utxos} set to use for the single transaction that funds all the channels.

multiwithdraw(*outputs*, *feerate=None*, *minconf=None*, *utxos=None*, ***kwargs*)
Send to {outputs} via Bitcoin transaction. Only select outputs with {minconf} confirmations.

newaddr(*addresstype=None*)
Get a new address of type {addresstype} of the internal wallet.

pay(*bolt11*, *msatoshi=None*, *label=None*, *riskfactor=None*, *description=None*, *maxfeepercent=None*, *retry_for=None*, *maxdelay=None*, *exemptfee=None*)
Send payment specified by {bolt11} with {msatoshi} (ignored if {bolt11} has an amount), optional {label} and {riskfactor} (default 1.0).

paystatus(*bolt11=None*)
Detail status of attempts to pay {bolt11} or any.

ping(*peer_id*, *length=128*, *pongbytes=128*)
Send {peer_id} a ping of length {len} asking for {pongbytes}.

plugin_list()
Lists all plugins lightningd knows about.

plugin_start(*plugin*)
Adds a plugin to lightningd.

plugin_startdir(*directory*)
Adds all plugins from a directory to lightningd.

plugin_stop(*plugin*)
Stops a lightningd plugin, will fail if plugin is not dynamic.

reserveinputs(*psbt*, *exclusive=True*)
Reserve any inputs in this psbt.

sendpay(*route*, *payment_hash*, **args*, ***kwargs*)
Send along {route} in return for preimage of {payment_hash}.

sendpsbt(*psbt*)
Finalize extract and broadcast a PSBT

setchannelfee(*id*, *base=None*, *ppm=None*)
Set routing fees for a channel/peer {id} (or 'all'). {base} is a value in millisatoshi that is added as base fee to any routed payment. {ppm} is a value added proportionally per-millionths to any routed payment volume in satoshi.

signmessage(*message*)
Sign a message with this node's secret key.

signpsbt(*psbt*, *signonly=None*)
Add internal wallet's signatures to PSBT

stop()
Shut down the lightningd process.

txdiscard(*txid*)
Cancel a Bitcoin transaction returned from txprepare. The outputs it was spending are released for other use.

txprepare(**args*, ***kwargs*)
Prepare a Bitcoin transaction which sends to [outputs]. The format of output is like [{address1: amount1}, {address2: amount2}], or [{address: "all"}]). Only select outputs with {minconf} confirmations.

Outputs will be reserved until you call txdiscard or txsend, or lightningd restarts.

txsend(*txid*)
Sign and broadcast a Bitcoin transaction returned from txprepare.

unreserveinputs(*psbt*)
Unreserve (or reduce reservation) on any UTXOs in this psbt were previously reserved.

utxopsbt(*satoshi*, *feerate*, *startweight*, *utxos*, *reserve=True*, *reservedok=False*, *locktime=None*)
Create a PSBT with given inputs, to give an output of satoshi.

waitanyinvoice(*lastpay_index=None*, *timeout=None*, ***kwargs*)
Wait for the next invoice to be paid, after {lastpay_index} (if supplied). Fail after {timeout} seconds has passed without an invoice being paid.

waitblockheight(*blockheight*, *timeout=None*)
Wait for the blockchain to reach the specified block height.

waitinvoice(*label*)
Wait for an incoming payment matching the invoice with {label}.

waitsendpay(*payment_hash*, *timeout=None*, *partid=None*)
Wait for payment for preimage of {payment_hash} to complete.

withdraw(*destination*, *satoshi*, *feerate=None*, *minconf=None*, *utxos=None*)
Send to {destination} address {satoshi} (or "all") amount via Bitcoin transaction. Only select outputs with {minconf} confirmations.

class pyln.client.**Plugin**(*stdout: Optional[io.TextIOBase] = None*, *stdin: Optional[io.TextIOBase] = None*, *autopatch: bool = True*, *dynamic: bool = True*, *init_features: Union[int, str, bytes, None] = None*, *node_features: Union[int, str, bytes, None] = None*, *invoice_features: Union[int, str, bytes, None] = None*)

Controls interactions with lightningd, and bundles functionality.

The Plugin class serves two purposes: it collects RPC methods and options, and offers a control loop that dispatches incoming RPC calls and hooks.

add_flag_option(*name: str*, *description: str*, *deprecated: bool = False*) → None
Add a flag option that we'd like to register with lightningd.

Needs to be called before *Plugin.run*, otherwise we might not end up getting it set.

add_hook(*name: str, func: Callable[[...], Union[str, int, float, bool, None, Dict[str, Any], List[Any]]], background: bool = False*) → None
Register a hook that is called synchronously by lightningd on events

add_method(*name: str, func: Callable[[...], Any], background: bool = False, category: Optional[str] = None, desc: Optional[str] = None, long_desc: Optional[str] = None, deprecated: bool = False*) → None
Add a plugin method to the dispatch table.

The function will be expected at call time (see *_dispatch*) and the parameters from the JSON-RPC call will be mapped to the function arguments. In addition to the parameters passed from the JSON-RPC call we add a few that may be useful:

- *plugin*: gets a reference to this plugin.
- *request*: gets a reference to the raw request as a dict. This corresponds to the JSON-RPC message that is being dispatched.

Notice that due to the python binding logic we may be mapping the arguments wrongly if we inject the plugin and/or request in combination with positional binding. To prevent issues the plugin and request argument should always be the last two arguments and have a default on None.

The *background* argument can be used to specify whether the method is going to return a result that should be sent back to the lightning daemon (*background=False*) or whether the method will return without sending back a result. In the latter case the method MUST use *request.set_result* or *result.set_exception* to return a result or raise an exception for the call.

The *category* argument can be used to specify the category of the newly created rpc command.

deprecated means that it won't appear unless *allow-deprecated-apis* is true (the default).

add_option(*name: str, default: Optional[str], description: Optional[str], opt_type: str = 'string', deprecated: bool = False*) → None
Add an option that we'd like to register with lightningd.

Needs to be called before *Plugin.run*, otherwise we might not end up getting it set.

add_subscription(*topic: str, func: Callable[[...], None]*) → None
Add a subscription to our list of subscriptions.

A subscription is an association between a topic and a handler function. Adding a subscription means that we will automatically subscribe to events from that topic with *lightningd* and, upon receiving a matching notification, we will call the associated handler. Notice that in order for the automatic subscriptions to work, the handlers need to be registered before we send our manifest, hence before *Plugin.run* is called.

async_hook(*method_name: str*) → Callable[[...], Callable[[...], None]]
Decorator to add an async plugin hook to the dispatch table.

Internally uses add_hook.

async_method(*method_name: str*, *category: Optional[str] = None*, *desc: Optional[str] = None*, *long_desc: Optional[str] = None*, *deprecated: bool = False*) → Callable[[...], Callable[[...], None]]
Decorator to add an async plugin method to the dispatch table.

Internally uses add_method.

hook(*method_name: str*) → Callable[[...], Callable[[...], Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]
Decorator to add a plugin hook to the dispatch table.

Internally uses add_hook.

init() → Callable[[...], Callable[[...], None]]
Decorator to add a function called after plugin initialization

method(*method_name: str*, *category: Optional[str] = None*, *desc: Optional[str] = None*, *long_desc: Optional[str] = None*, *deprecated: bool = False*) → Callable[[...], Callable[[...], Union[str, int, float, bool, None, Dict[str, Any], List[Any]]]]
Decorator to add a plugin method to the dispatch table.

Internally uses add_method.

subscribe(*topic: str*) → Callable[[...], Callable[[...], None]]
Function decorator to register a notification handler.

exception pyln.client.**RpcError**(*method: str*, *payload: dict*, *error: str*)

class pyln.client.**Millisatoshi**(*v: Union[int, str, decimal.Decimal]*)
A subtype to represent thousandths of a satoshi.

Many JSON API fields are expressed in millisatoshis: these automatically get turned into Millisatoshi types. Converts to and from int.

to_approx_str(*digits: int = 3*) → str
Returns the shortmost string using common units representation.

Rounds to significant *digits*. Default: 3

to_btc() → decimal.Decimal
Return a Decimal representing the number of bitcoin.

to_btc_str() → str
Return a string of form 12.34567890btc or 12.34567890123btc.

to_satoshi() → decimal.Decimal
Return a Decimal representing the number of satoshis.

to_satoshi_str() → str
Return a string of form 1234sat or 1234.567sat.

pyln.client.**monkey_patch**(*plugin: pyln.client.plugin.Plugin*, *stdout: bool = True*, *stderr: bool = False*) → None
Monkey patch stderr and stdout so we use notifications instead.

A plugin commonly communicates with lightningd over its stdout and stdin filedescriptor, so if we use them in some other way (printing, logging, ...) we're breaking our communication channel. This function monkey patches these streams in the *sys* module to be redirected to a *PluginStream* which wraps anything that would get written to these streams into valid log notifications that can be interpreted and printed by *lightningd*.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

M

N

P

R

S

T

U

W