ElectrumX Documentation

Release ElectrumX 1.15.0

Neil Booth

Jul 10, 2020

Contents

A reimplementation of Electrum-Server for a future with bigger blocks.

The current version is ElectrumX 1.15.0.

CHAPTER 1

Source Code

The project is hosted on GitHub. and uses Travis for Continuous Integration.

Please submit an issue on the bug tracker if you have found a bug or have a suggestion to improve the server.

CHAPTER 2

Authors and License

Neil Booth wrote the vast majority of the code; see *Authors*. Python version at least 3.7 is required.

The code is released under the MIT Licence.

CHAPTER 3

Getting Started

See *HOWTO*.

There is also an installer available that simplifies the installation on various Linux-based distributions, and a Dockerfile available .

CHAPTER 4

Documentation

4.1 Features

- Efficient, lightweight reimplementation of electrum-server
- Fast synchronization of bitcoin mainnet from Genesis. Recent hardware should synchronize in well under 24 hours. The fastest time to height 448k (mid January 2017) reported is under 4h 30m. On the same hardware JElectrum would take around 4 days and electrum-server probably around 1 month.
- Various configurable means of controlling resource consumption and handling bad clients and denial of service attacks. These include maximum connection counts, subscription limits per-connection and across all connections, maximum response size, per-session bandwidth limits, and session timeouts.
- Minimal resource usage once caught up and serving clients; tracking the transaction mempool appears to be the most expensive part.
- Mostly asynchronous processing of new blocks, mempool updates, and client requests. Busy clients should not noticeably impede other clients' requests and notifications, nor the processing of incoming blocks and mempool updates.
- Daemon failover. More than one daemon can be specified, and ElectrumX will failover round-robin style if the current one fails for any reason.
- Peer discovery protocol removes need for IRC
- Coin abstraction makes compatible altcoin and testnet support easy.

4.2 Implementation

ElectrumX does not do any pruning or throwing away of history. I want to retain this property for as long as it is feasible, and it appears efficiently achievable for the foreseeable future with plain Python.

The following all play a part in making it efficient as a Python blockchain indexer:

- aggressive caching and batching of DB writes

- more compact and efficient representation of UTXOs, address index, and history. Electrum Server stores full transaction hash and height for each UTXO, and does the same in its pruned history. In contrast ElectrumX just stores the transaction number in the linear history of transactions. For at least another 5 years this transaction number will fit in a 4-byte integer, and when necessary expanding to 5 or 6 bytes is trivial. ElectrumX can determine block height from a simple binary search of tx counts stored on disk. ElectrumX stores historical transaction hashes in a linear array on disk.
- placing static append-only metadata indexable by position on disk rather than in levelDB. It would be nice to do this for histories but I cannot think of a way.
- avoiding unnecessary or redundant computations, such as converting address hashes to human-readable ASCII strings with expensive bignum arithmetic, and then back again.
- better choice of Python data structures giving lower memory usage as well as faster traversal
- leveraging asyncio for asynchronous prefetch of blocks to mostly eliminate CPU idling. As a Python program ElectrumX is unavoidably single-threaded in its essence; we must keep that CPU core busy.

Python's `asyncio` means ElectrumX has no (direct) use for threads and associated complications.

4.3 Roadmap

- break ElectrumX up into simple services that initially can be run in separate processes on a single host. Then support running them on different hosts, and finally support sharding. With this we can take advantage of multiple cores and hosts, and scale to much larger block sizes. This should solve several issues with the current history storage mechanism.
- fully asynchronous operation. At present too much is synchronous, such as file system access.
- protocol improvements targeting better client and server scalability to large wallets (100k addresses) and address histories. Some aspects of the current protocol are very inefficient.
- investigate speaking the Bitcoin protocol and connecting to the Bitcoin network directly for some queries. This could lead to ElectrumX being runnable with a node without a tx index, or a pruning node, or not needing to run a node at all. ElectrumX would store all blocks itself and index the transactions therein.
- lifting internal limits such as maximum 4 billion transactions
- supporting better user privacy. I believe significantly improved user address privacy might be possible with a simple addition to the protocol, and assuming a server network of which a reasonable fraction (40%?) are cooperative and non-colluding
- new features such as possibly adding label or wallet server functionality

4.4 ChangeLog

Note: Version 1.15.0 will be the final ElectrumX release with altcoin support, future releases will be Bitcoin-only. ElectrumX needs to scale quickly and support for various other coins and their idiosyncracies is distracting and unhelpful. Anyone wishing to maintain an ElectrumX repository with altcoin support is free to do so as explained in the licence.

Note: It is strongly recommended you upgrade to Python 3.7, which fixes bugs in asyncio that caused an ever-growing open file count and memory consumption whilst serving clients. Those problems should not occur with Python 3.7.

4.4.1 Version 1.15.0 (27 May 2020)

- switch to 5-byte txnums to handle larger blockchains. Upgrade DBs during restart.
- accurate clearing of stale caches
- coin additions / updates: NavCoin + Hush + VersusCoin + Zero (cipig), DashRegtest (colmenero), Quebecoin (morinpa), Primecoin (Sunny King), multiple (Panagiotis David), RVN (standard-error), Sumcoin
- other: Jeremy Rand, Jin Eguchi, ddude, Jonathan Cross, Carsen Klock, cipig

4.4.2 Version 1.14.0 (19 Jan 2020)

- require Python 3.7
- support for Bitcoin SV Genesis activation
- DB upgrade to allow for larger transactions. Your DB will automatically upgrade when starting, the upgrade should take approximately 15 mintues.
- fix server shutdown process
- fix cache race condition (issue #909)
- faster initial sync
- coin additions / updates: Emercoin (yakimka), Feathercoin (wellenreiter01), Peercoin (peerchemist), Namecoin (JeremyRand), Zcoin (a-bezrukov), Simplicity, Mice (ComputerCraftr), Sibcoin testnet (TriKriSta), Odin (Manbearpixel),
- other: h2o10, osagga, Sombernight, breign, pedr0-fr, wingsuit

4.4.3 Version 1.13.0 (26 Sep 2019)

- daemon: use a single connection for all requests rather than a connection per request. Distinguish handling of JSON and HTTP errors
- recognise OP_FALSE OP_RETURN scripts as unspendable
- peers - attempt to bind to correct local IP address
- improve name support (domob1812)
- coin additions / updates: BitZeny (y-chan), ZCoin (a-bezrukov), Emercoin (yakimka), BSV (Roger Taylor), Bellcoin (streetcrypto7), Ritocoin (traysi), BTC (Sombernight), PIVX (mrcarlanthony), Monacoin (wakiyamap)), NamecoinRegtest (JeremyRand), Axe (ddude1), Xaya (domob1812), GZRO (MrNaif2018), Ravencoin (standard-error)
- other: gits7r

4.4.4 Version 1.12.0 (13 May 2019)

- require aiorpcX 0.18.1. This introduces websocket support. The environment variables changed accordingly; see *SERVICES* and *REPORT_SERVICES*.
- work around bug in recent versions of uvloop
- aiorpcX upgrade fixes from Shane M
- coin additions / updates: BitcoinSV, Bolivarcoin (Jose Luis Estevez), BTC Testnet (ghost43), Odin (Pixxl)

4.4.5 Version 1.11.0 (18 Apr 2019)

- require aiorpcX 0.15.x
- require aiohttp 3.3 or higher; earlier versions had a problematic bug
- add *REQUEST_TIMEOUT* and *LOG_LEVEL* environment variables
- mark 4 old environment variables obsolete. ElectrumX won't start until they are removed
- getinfo local RPC cleaned up and shows more stats
- miscellaneous fixes and improvements
- more efficient handling of some RPC methods, particularly *blockchain.transaction.get_merkle()*
- coin additions / updates: BitcoinSV scaling testnet (Roger Taylor), Dash (zebra lucky),
- issues resolved: #566, #731, #795

4.4.6 Version 1.10.1 (13 Apr 2019)

- introduce per-request costing. See environment variables documentation for new variables *COST_SOFT_LIMIT*, *COST_HARD_LIMIT*, *REQUEST_SLEEP*, *INITIAL_CONCURRENT*, *BANDWIDTH_UNIT_COST*. Sessions are placed in groups with which they share some of their costs. Prior cost is remembered across reconnects.
- require aiorpcX 0.13.5 for better concurrency handling
- require clients use protocol 1.4 or higher
- handle transaction.get_merkle requests more efficiently (ghost43)
- Windows support (sancoder)
- peers improvements (ghost43)
- report mempool and block sizes in logs
- electrumx_rpc: timeout raised to 30s, fix session request counts
- other tweaks and improvements by Bjorge Dijkstra, ghost43, peleion,
- coin additions / updates: ECA (Jenova7), ECCoin (smogm), GXX (DEVCØN), BZX (2INFINITY), DeepOnion (Liam Alford), CivX / EXOS (turcol)

4.4.7 Version 1.10.0 (15 Mar 2019)

- extra countermeasures to limit BTC phishing effectiveness (ghost43)
- peers: mark blacklisted peers bad; force retry blacklisted peers (ghost43)
- coin additions / updates: Monacoin (wakiyamap), Sparks (Mircea Rila), ColossusXT, Polis, MNPCoin, Zcoin, GINCoin (cronos), Grosetlcoin (gruve-p), Dash (konez2k), Bitsend (David), Ravencoin (standard-error), Onixcoin (Jose Estevez), SnowGem
- coin removals: Gobyte, Moneci (cronos)
- minor tweaks by d42
- issues fixed #660 - unclean shutdowns during initial sync

4.4.8 Version 1.9.5 (08 Feb 2019)

- server blacklist logic (ecdsa)
- require aiorpcX 0.10.4
- remove dead wallet code
- fix #727 - not listing same peer twice

4.4.9 Version 1.9.4 (07 Feb 2019)

- require aiorpcX 0.10.3
- fix #713

4.4.10 Version 1.9.3 (05 Feb 2019)

- ignore potential sybil peers
- coin additions / updates: BitcoinCashABC (cculianu), Monacoin (wakiyamap)

4.4.11 Version 1.9.2 (03 Feb 2019)

- restore protocol version 1.2 and send a warning for old BTC Electrum clients that they need to upgrade. This is an attempt to protect users of old versions of Electrum from the ongoing phishing attacks
- increase default MAX_SEND for AuxPow Chains. Truncate AuxPow for block heights covered by a checkpoint. (jeremyrand)
- coin additions / updates: NMC (jeremyrand), Dash (zebra-lucky), PeerCoin (peerchemist), BCH testnet (Mark Lundeberg), Unitus (ChekaZ)
- tighter RPC param checking (ghost43)

4.4.12 Version 1.9.1 (11 Jan 2019)

- fix #684

4.4.13 Version 1.9.0 (10 Jan 2019)

- minimum protocol version is now 1.4
- coin additions / updates: BitcoinSV, SmartCash (rc125), NIX (phamels), Minexcoin (joesixpack), BitcoinABC (mblunderburg), Dash (zebra-lucky), BitcoinABCRegtest (ezegom), AXE (slowdive), NOR (flo071), Bitcoin-Plus (bushsolo), Myriadcoin (cryptapus), Trezarcoin (ChekaZ), Bitcoin Diamond (John Shine),
- close #554, #653, #655
- other minor tweaks (Michael Schmoock, Michael Taborsky)

Neil Booth kyuupichan@gmail.com https://github.com/kyuupichan

4.5 HOWTO

4.5.1 Prerequisites

ElectrumX should run on any flavour of unix. I have run it successfully on MacOS and DragonFlyBSD. It won't run out-of-the-box on Windows, but the changes required to make it do so should be small - pull requests are welcome.

Package	Notes
Python3	ElectrumX uses asyncio. Python version >= 3.7 is **required**.
aiohttp	Python library for asynchronous HTTP. Version >= 2.0 required.
pylru	Python LRU cache package.
DB Engine	A database engine package is required; two are supported (see *Database Engine* below).

Some coins need an additional package, typically for their block hash functions. For example, x11_hash is required for DASH.

You **must** to be running a non-pruning bitcoin daemon with:

```
txindex=1
```

set in its configuration file. If you have an existing installation of bitcoind and have not previously set this you will need to reindex the blockchain with:

```
bitcoind -reindex
```

which can take some time.

While not a requirement for running ElectrumX, it is intended to be run with supervisor software such as Daniel Bernstein's daemontools, Gerrit Pape's runit package or **`systemd`**. These make administration of secure unix servers very easy, and I strongly recommend you install one of these and familiarise yourself with them. The instructions below and sample run scripts assume `daemontools`; adapting to `runit` should be trivial for someone used to either.

When building the database from the genesis block, ElectrumX has to flush large quantities of data to disk and its DB. You will have a better experience if the database directory is on an SSD than on an HDD. Currently to around height 611,600 of the Bitcoin blockchain the final size of the leveldb database, and other ElectrumX file metadata comes to just over 46.9GB (43.7 GiB). LevelDB needs a bit more for brief periods, and the block chain is only getting longer, so I would recommend having at least 70-80GB of free space before starting.

4.5.2 Database Engine

You can choose from LevelDB and RocksDB to store transaction information on disk. The time taken and DB size is not significantly different. We tried to support LMDB but its history write performance was much worse.

You will need to install one of:

- plyvel for LevelDB
- python-rocksdb for RocksDB (*pip3 install python-rocksdb*)
- pyrocksdb for an unmaintained version that doesn't work with recent releases of RocksDB

4.5.3 Running

Install the prerequisites above.

Check out the code from Github:

```
git clone https://github.com/kyuupichan/electrumx.git
cd electrumx
```

You can install with `setup.py` or run the code from the source tree or a copy of it.

You should create a standard user account to run the server under; your own is probably adequate unless paranoid. The paranoid might also want to create another user account for the daemontools logging process. The sample scripts and these instructions assume it is all under one account which I have called `electrumx`.

Next create a directory where the database will be stored and make it writeable by the `electrumx` account. I recommend this directory live on an SSD:

```
mkdir /path/to/db_directory
chown electrumx /path/to/db_directory
```

Process limits

You must ensure the ElectrumX process has a large open file limit. During sync it should not need more than about 1,024 open files. When serving it will use approximately 256 for LevelDB plus the number of incoming connections. It is not unusual to have 1,000 to 2,000 connections being served, so I suggest you set your open files limit to at least 2,500.

Note that setting the limit in your shell does *NOT* affect ElectrumX unless you are invoking ElectrumX directly from your shell. If you are using **`systemd`**, you need to set it in the `.service` file (see contrib/systemd/electrumx.service).

Using daemontools

Next create a daemontools service directory; this only holds symlinks (see daemontools documentation). The **`svscan`** program will ensure the servers in the directory are running by launching a **`supervise`** supervisor for the server and another for its logging process. You can run **`svscan`** under the *electrumx* account if that is the only one involved (server and logger) otherwise it will need to run as root so that the user can be switched to electrumx.

Assuming this directory is called `service`, you would do one of:

```
mkdir /service        # If running svscan as root
mkdir ~/service       # As electrumx if running svscan as that a/c
```

Next create a directory to hold the scripts that the **`supervise`** process spawned by **`svscan`** will run - this directory must be readable by the **`svscan`** process. Suppose this directory is called `scripts`, you might do:

```
mkdir -p ~/scripts/electrumx
```

Then copy the all sample scripts from the ElectrumX source tree there:

```
cp -R /path/to/repo/electrumx/contrib/daemontools ~/scripts/electrumx
```

This copies 3 things: the top level server run script, a `log/` directory with the logger **`run`** script, an `env/` directory.

You need to configure the *environment variables* under `env/` to your setup. ElectrumX server currently takes no command line arguments; all of its configuration is taken from its environment which is set up according to `env/`

directory (see *envdir* man page). Finally you need to change the **log/run** script to use the directory where you want the logs to be written by multilog. The directory need not exist as **multilog** will create it, but its parent directory must exist.

Now start the **svscan** process. This will not do much as the service directory is still empty:

```
svscan ~/service & disown
```

svscan is now waiting for services to be added to the directory:

```
cd ~/service
ln -s ~/scripts/electrumx electrumx
```

Creating the symlink will kick off the server process almost immediately. You can see its logs with:

```
tail -F /path/to/log/dir/current | tai64nlocal
```

Using systemd

This repository contains a sample systemd unit file that you can use to setup ElectrumX with systemd. Simply copy it to `/etc/systemd/system`:

```
cp contrib/systemd/electrumx.service /etc/systemd/system/
```

The sample unit file assumes that the repository is located at `/home/electrumx/electrumx`. If that differs on your system, you need to change the unit file accordingly.

You need to set a few *environment variables* in `/etc/electrumx.conf`.

Now you can start ElectrumX using **systemctl**:

```
systemctl start electrumx
```

You can use **journalctl** to check the log output:

```
journalctl -u electrumx -f
```

Once configured you may want to start ElectrumX at boot:

```
systemctl enable electrumx
```

Warning: systemd is aggressive in forcibly shutting down processes. Depending on your hardware, ElectrumX can need several minutes to flush cached data to disk during initial sync. You should set TimeoutStopSec to *at least* 10 mins in your `.service` file.

Installing on Raspberry Pi 3

To install on the Raspberry Pi 3 you will need to update to the `stretch` distribution. See the full procedure in contrib/raspberrypi3/install_electrumx.sh.

See also contrib/raspberrypi3/run_electrumx.sh for an easy way to configure and launch electrumx.

4.5.4 Sync Progress

Time taken to index the blockchain depends on your hardware of course. As Python is single-threaded most of the time only 1 core is kept busy. ElectrumX uses Python's `asyncio` to prefill a cache of future blocks asynchronously to keep the CPU busy processing the chain without pausing.

Consequently there will probably be only a minor boost in performance if the daemon is on the same host. It may even be beneficial to have the daemon on a *separate* machine so the machine doing the indexing has its caches and disk I/O tuned to that task only.

The *CACHE_MB* environment variable controls the total cache size ElectrumX uses; see *here* for caveats.

Here is my experience with the codebase of early 2017 (the current codebase is faster), to given heights and rough wall-time. The period from heights 363,000 to 378,000 is the most sluggish:

```
                Machine A     Machine B
181,000           25m 00s      5m 30s
283,500                        1h 00m
321,800                        1h 40m
357,000           12h 32m      2h 41m
386,000           21h 56m      4h 25m
414,200         1d 12h 29m      6h 30m
447,168         2d 13h 20m      9h 47m
```

Machine A: a low-spec 2011 1.6GHz AMD E-350 dual-core fanless CPU, 8GB RAM and a DragonFlyBSD UFS filesystem on an SSD. It requests blocks over the LAN from a bitcoind on machine B. `DB_CACHE` the default of 1,200. LevelDB.

Machine B: a late 2012 iMac running Sierra 10.12.2, 2.9GHz quad-core Intel i5 CPU with an HDD and 24GB RAM. Running bitcoind on the same machine. `DB_CACHE` set to 1,800. LevelDB.

For chains other than bitcoin-mainnet synchronization should be much faster.

Note: ElectrumX will not serve normal client connections until it has fully synchronized and caught up with your daemon. However LocalRPC connections are served at all times.

4.5.5 Terminating ElectrumX

The preferred way to terminate the server process is to send it the `stop` RPC command:

```
electrumx_rpc stop
```

or alternatively on Unix the `INT` or `TERM` signals. For a daemontools supervised process this can be done by bringing it down like so:

```
svc -d ~/service/electrumx
```

ElectrumX will note receipt of the signals in the logs, and ensure the block chain index is flushed to disk before terminating. You should be patient as flushing data to disk can take many minutes.

ElectrumX uses the transaction functionality, with fsync enabled, of the databases. I have written it with the intent that, to the extent the atomicity guarantees are upheld by the DB software, the operating system, and the hardware, the database should not get corrupted even if the ElectrumX process if forcibly killed or there is loss of power. The worst case should be having to restart indexing from the most recent UTXO flush.

Once the process has terminated, you can start it up again with:

```
svc -u ~/service/electrumx
```

You can see the status of a running service with:

```
svstat ~/service/electrumx
```

svscan can of course handle multiple services simultaneously from the same service directory, such as a testnet or altcoin server. See the man pages of these various commands for more information.

4.5.6 Understanding the Logs

You can see the logs usefully like so:

```
tail -F /path/to/log/dir/current | tai64nlocal
```

Here is typical log output on startup:

```
INFO:BlockProcessor:switching current directory to /crucial/server-good
INFO:BlockProcessor:using leveldb for DB backend
INFO:BlockProcessor:created new database
INFO:BlockProcessor:creating metadata diretcory
INFO:BlockProcessor:software version: ElectrumX 0.10.2
INFO:BlockProcessor:DB version: 5
INFO:BlockProcessor:coin: Bitcoin
INFO:BlockProcessor:network: mainnet
INFO:BlockProcessor:height: -1
INFO:BlockProcessor:tip:␣
↪0000000000000000000000000000000000000000000000000000000000000000
INFO:BlockProcessor:tx count: 0
INFO:BlockProcessor:sync time so far: 0d 00h 00m 00s
INFO:BlockProcessor:reorg limit is 200 blocks
INFO:Daemon:daemon at 192.168.0.2:8332/
INFO:BlockProcessor:flushing DB cache at 1,200 MB
INFO:Controller:RPC server listening on localhost:8000
INFO:Prefetcher:catching up to daemon height 447,187...
INFO:Prefetcher:verified genesis block with hash␣
↪000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
INFO:BlockProcessor:our height: 9 daemon: 447,187 UTXOs 0MB hist 0MB
INFO:BlockProcessor:our height: 52,509 daemon: 447,187 UTXOs 9MB hist 14MB
INFO:BlockProcessor:our height: 85,009 daemon: 447,187 UTXOs 12MB hist 31MB
INFO:BlockProcessor:our height: 102,384 daemon: 447,187 UTXOs 15MB hist 47MB
[...]
INFO:BlockProcessor:our height: 133,375 daemon: 447,187 UTXOs 80MB hist 222MB
INFO:BlockProcessor:our height: 134,692 daemon: 447,187 UTXOs 96MB hist 250MB
INFO:BlockProcessor:flushed to FS in 0.7s
INFO:BlockProcessor:flushed history in 16.3s for 1,124,512 addrs
INFO:BlockProcessor:flush #1 took 18.7s.  Height 134,692 txs: 941,963
INFO:BlockProcessor:tx/sec since genesis: 2,399, since last flush: 2,400
INFO:BlockProcessor:sync time: 0d 00h 06m 32s  ETA: 1d 13h 03m 42s
```

Under normal operation these cache stats repeat once or twice a minute. UTXO flushes can take several minutes and look like this:

```
INFO:BlockProcessor:our height: 378,745 daemon: 447,332 UTXOs 1,013MB hist 184MB
INFO:BlockProcessor:our height: 378,787 daemon: 447,332 UTXOs 1,014MB hist 194MB
INFO:BlockProcessor:flushed to FS in 0.3s
```

(continues on next page)

(continued from previous page)

```
INFO:BlockProcessor:flushed history in 13.4s for 934,933 addrs
INFO:BlockProcessor:flushed 6,403 blocks with 5,879,440 txs, 2,920,524 UTXO adds, 3,
↪646,572 spends in 93.1s, committing...
INFO:BlockProcessor:flush #120 took 226.4s.  Height 378,787 txs: 87,695,588
INFO:BlockProcessor:tx/sec since genesis: 1,280, since last flush: 359
INFO:BlockProcessor:sync t ime: 0d 19h 01m 06s  ETA: 3d 21h 17m 52s
INFO:BlockProcessor:our height: 378,812 daemon: 447,334 UTXOs 10MB hist 10MB
```

The ETA shown is just a rough guide and in the short term can be quite volatile. It tends to be a little optimistic at first; once you get to height 280,000 is should be fairly accurate.

4.5.7 Creating a self-signed SSL certificate

These instructions are based on those of the `electrum-server` documentation.

To run an SSL server you need to generate a self-signed certificate using openssl. Alternatively you could not set `SSL_PORT` in the environment and not serve over SSL, but this is not recommended.

Use the sample code below to create a self-signed cert with a recommended validity of 5 years. You may supply any information for your sign request to identify your server. They are not currently checked by the client except for the validity date. When asked for a challenge password just leave it empty and press enter:

```
$ openssl genrsa -out server.key 2048
$ openssl req -new -key server.key -out server.csr
...
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Common Name (eg, YOUR name) []: electrum-server.tld
...
A challenge password []:
...
$ openssl x509 -req -days 1825 -in server.csr -signkey server.key -out server.crt
```

The `server.crt` file goes in *`SSL_CERTFILE`* and `server.key` in *`SSL_KEYFILE`* in the server process's environment.

Starting with Electrum 1.9, the client will learn and locally cache the SSL certificate for your server upon the first request to prevent man-in-the middle attacks for all further connections.

If your certificate is lost or expires on the server side, you will need to run your server with a different server name and a new certificate. Therefore it's a good idea to make an offline backup copy of your certificate and key in case you need to restore them.

4.5.8 Running on a privileged port

You may choose to run electrumx on a different port than 50001 / 50002. If you choose a privileged port (< 1024) it makes sense to make use of a iptables NAT rule.

An example, which will forward Port 110 to the internal port 50002 follows:

```
iptables -t nat -A PREROUTING -p tcp --dport 110 -j DNAT --to-destination 127.0.0.
↪1:50002
```

You can then set the port as follows and advertise the service externally on the privileged port:

```
REPORT_SSL_PORT=110
```

4.6 Environment Variables

ElectrumX takes no command line arguments, instead its behaviour is controlled by environment variables. Only a few are required to be given, the rest will have sensible defaults if not specified. Many of the defaults around resource usage are conservative; I encourage you to review them.

Note: set *SERVICES* appropriately to be able to connect to your server. For clients across the internet to know what services you offer you must advertize your services with *REPORT_SERVICES*.

4.6.1 Required

These environment variables are always required:

COIN
: Must be a NAME from one of the Coin classes in lib/coins.py.

DB_DIRECTORY
: The path to the database directory. Relative paths should be relative to the parent process working directory. This is the directory of the *run* script if you use it.

DAEMON_URL
: A comma-separated list of daemon URLs. If more than one is provided ElectrumX will initially connect to the first, and failover to subsequent ones round-robin style if one stops working.

  The generic form of a daemon URL is:

  ```
  http://username:password@hostname:port/
  ```

  The leading `http://` is optional, as is the trailing slash. The `:port` part is also optional and will default to the standard RPC port for *COIN* and *NET* if omitted.

Note: With the above set your server will run and index the chain. To enable incoming connections you must set *SERVICES*, and for others to be aware of your server set *REPORT_SERVICES*.

4.6.2 For the `run` script

The following are required if you use the `run` script:

ELECTRUMX
: The path to the electrumx_server script. Relative paths should be relative to the directory of the `run` script.

USERNAME
: The username the server will run as.

4.6.3 Services

These two environment variables are comma-separated lists of individual *services*.

A **service** has the general form:

```
protocol://host:port
```

protocol is case-insensitive. The recognised protocols are:

```
tcp    Plaintext TCP sockets
ssl    SSL-encrypted TCP sockets
ws     Plaintext websockets
wss    SSL-encrypted websockets
rpc    Plaintext RPC
```

In a services list, a protocol can be specified multiple times, with different hosts or ports. This might be useful for multi-homed hosts, or if you offer both Tor and clearnet services.

host can be a hostname, an IPv4 address, or an IPv6 address enclosed in square brackets.

port is an integer from `1` to `65535` inclusive.

Where documented, one or more of *protocol*, *host* and *port* can be omitted, in which case a default value will be assumed.

Here are some examples of valid services:

```
tcp://host.domain.tld:50001           # Hostname, lowercase protocol, port
SSL://23.45.67.78:50002               # An IPv4 address, upper-case protocol, port
rpC://localhost                       # Host as a string, mixed-case protocol,␣
↪default port
ws://[1234:5678:abcd::5601]:8000      # Host as an IPv6 address
wss://h3ubaasdlkheryasd.onion:50001   # Host as a Tor ".onion" address
rpc://:8000                           # Default host, port given
host.domain.tld:5151                  # Default protocol, hostname, port
rpc://                                # RPC protocol, default host and port
```

Note: ElectrumX will not serve any incoming connections until it has fully caught up with your bitcoin daemon. The only exception is local **RPC** connections, which are served at any time after the server has initialized.

SERVICES

A comma-separated list of services ElectrumX will accept incoming connections for.

This environment variable determines what interfaces and ports the server listens on, so must be set correctly for any connection to the server to succeed. If unset or empty, ElectrumX will not listen for any incoming connections.

protocol can be any recognised protocol.

host defaults to all of the machine's interfaces, except if the protocol is **rpc**, when it defaults to `localhost`.

port can only be defaulted for **rpc** where the default is `8000`.

On most Unix systems ports below 1024 require elevated privileges so choosing a higher port is advisable. On Debian for example, this can be achieved by installing libcap2-bin package:

```
sudo apt-get update && sudo apt-get -y install libcap2-bin
sudo setcap cap_net_bind_service=+ep /path/to/electrumx_server
```

If any listed service has protocol **ssl** or **wss** then *SSL_CERTFILE* and *SSL_KEYFILE* must be defined.

Tor **onion** addresses are invalid in *SERVICES*.

Here is an example value of the *SERVICES* environment variable:

```
tcp://:50001,ssl://:50002,wss://:50004,rpc://
```

This serves **tcp**, **ssl**, **wss** on all interfaces on ports 50001, 50002 and 50004 respectively. **rpc** is served on its default host `localhost` and default port `8000`.

REPORT_SERVICES
A comma-separated list of services ElectrumX will advertize and other servers in the server network (if peer discovery is enabled), and any successful connection.

This environment variable must be set correctly, taking account of your network, firewall and router setup, for clients and other servers to see how to connect to your server. If not set or empty, no services are advertized.

The **rpc** protocol, special IP addresses (including private ones if peer discovery is enabled), and `localhost` are invalid in *REPORT_SERVICES*.

Here is an example value of the *REPORT_SERVICES* environment variable:

```
tcp://sv.usebsv.com:50001,ssl://sv.usebsv.com:50002,wss://sv.usebsv.com:50004
```

This advertizes **tcp**, **ssl**, **wss** services at `sv.usebsv.com` on ports 50001, 50002 and 50004 respectively.

Note: Certificate Authority-signed certificates don't work over Tor, so you should only have Tor services' in *REPORT_SERVICES* if yours is self-signed.

SSL_CERTFILE
The filesystem path to your SSL certificate file.

Creating a self-signed SSL certificate

SSL_KEYFILE
The filesystem path to your SSL key file.

Creating a self-signed SSL certificate

4.6.4 Miscellaneous

These environment variables are optional:

LOG_FORMAT
The Python logging format string to use. Defaults to `%(levelname)s:%(name)s:%(message)s`.

LOG_LEVEL
The default Python logging level, a case-insensitive string. Useful values are 'debug', 'info', 'warning' and 'error'.

ALLOW_ROOT
Set this environment variable to anything non-empty to allow running ElectrumX as root.

NET
Must be a *NET* from one of the **Coin** classes in lib/coins.py. Defaults to `mainnet`.

DB_ENGINE
Database engine for the UTXO and history database. The default is `leveldb`. The other alternative is

`rocksdb`. You will need to install the appropriate python package for your engine. The value is not case sensitive.

DONATION_ADDRESS
The server donation address reported to Electrum clients. Defaults to empty, which Electrum interprets as meaning there is none.

BANNER_FILE
The path to a banner file to serve to clients in Electrum's "Console" tab. Relative file paths must be relative to *DB_DIRECTORY*. The banner file is re-read for each new client.

You can place several meta-variables in your banner file, which will be replaced before serving to a client.

- `$SERVER_VERSION` is replaced with the ElectrumX version you are running, such as `1.0.10`.
- `$SERVER_SUBVERSION` is replaced with the ElectrumX user agent string. For example, `ElectrumX 1.0.10`.
- `$DAEMON_VERSION` is replaced with the daemon's version as a dot-separated string. For example `0.12.1`.
- `$DAEMON_SUBVERSION` is replaced with the daemon's user agent string. For example, `/BitcoinUnlimited:0.12.1(EB16; AD4)/`.
- `$DONATION_ADDRESS` is replaced with the address from the *DONATION_ADDRESS* environment variable.

See here for a script that updates a banner file periodically with useful statistics about fees, last block time and height, etc.

TOR_BANNER_FILE
As for *BANNER_FILE* (which is also the default) but shown to incoming connections believed to be to your Tor hidden service.

ANON_LOGS
Set to anything non-empty to replace IP addresses in logs with redacted text like `xx.xx.xx.xx:xxx`. By default IP addresses will be written to logs.

LOG_SESSIONS
The number of seconds between printing session statistics to the log. The output is identical to the *sessions* RPC command except that *ANON_LOGS* is honoured. Defaults to 3600. Set to zero to suppress this logging.

REORG_LIMIT
The maximum number of blocks to be able to handle in a chain reorganisation. ElectrumX retains some fairly compact undo information for this many blocks in levelDB. The default is a function of *COIN* and *NET*; for Bitcoin mainnet it is 200.

EVENT_LOOP_POLICY
The name of an event loop policy to replace the default asyncio policy, if any. At present only `uvloop` is accepted, in which case you must have installed the uvloop Python package.

If you are not sure what this means leave it unset.

DROP_CLIENT
Set a regular expression to disconnect any client based on their version string. For example to drop versions from 1.0 to 1.2 use the regex `1\.[0-2]\.\d+`.

4.6.5 Resource Usage Limits

The following environment variables are all optional and help to limit server resource consumption and prevent simple DoS.

Address subscriptions in ElectrumX are very cheap - they consume about 160 bytes of memory each and are processed efficiently. I feel the two subscription-related defaults below are low and encourage you to raise them.

MAX_SESSIONS
The maximum number of incoming connections. Once reached, TCP and SSL listening sockets are closed until the session count drops naturally to 95% of the limit. Defaults to 1,000.

MAX_SEND
The maximum size of a response message to send over the wire, in bytes. Defaults to 1,000,000 (except for Aux-PoW coins, which default to 10,000,000). Values smaller than 350,000 are taken as 350,000 because standard Electrum protocol header "chunk" requests are almost that large.

The Electrum protocol has a flaw in that address histories must be served all at once or not at all, an obvious avenue for abuse. *MAX_SEND* is a stop-gap until the protocol is improved to admit incremental history requests. Each history entry is approximately 100 bytes so the default is equivalent to a history limit of around 10,000 entries, which should be ample for most legitimate users. If you use a higher default bear in mind one client can request history for multiple addresses. Also note that the largest raw transaction you will be able to serve to a client is just under half of *MAX_SEND*, as each raw byte becomes 2 hexadecimal ASCII characters on the wire. Very few transactions on Bitcoin mainnet are over 500KB in size.

COST_SOFT_LIMIT

COST_HARD_LIMIT

REQUEST_SLEEP

INITIAL_CONCURRENT
All values are integers. *COST_SOFT_LIMIT* defaults to `1,000`, *COST_HARD_LIMIT* to `10,000`, *REQUEST_SLEEP* to `2,500` milliseconds, and *INITIAL_CONCURRENT* to `10` concurrent requests.

The server prices each request made to it based upon an estimate of the resources needed to process it. Factors include whether the request uses bitcoind, how much bandwidth it uses, and how hard it hits the databases.

To set a base for the units, a *blockchain.scripthash.subscribe()* subscription to an address with a history of 2 or fewer transactions is costed at `1.0` before considering the bandwidth consumed. *server.ping()* is costed at `0.1`.

As the total cost of a session goes over the soft limit, its requests start to be throttled in two ways. First, the number of requests for that session that the server will process concurrently is reduced. Second, each request starts to sleep a little before being handled.

Before throttling starts, the server will process up to *INITIAL_CONCURRENT* requests concurrently without sleeping. As the session cost ranges from *COST_SOFT_LIMIT* to *COST_HARD_LIMIT*, concurrency drops linearly to zero and each request's sleep time increases linearly up to *REQUEST_SLEEP* milliseconds. Once the hard limit is reached, the session is disconnected.

In order that non-abusive sessions can continue to be served, a session's cost gradually decays over time. Subscriptions have an ongoing servicing cost, so the decay is slower as the number of subscriptions increases.

If a session disconnects, ElectrumX continues to associate its cost with its IP address, so if it immediately reconnects it will re-acquire its previous cost allocation.

A server operator should experiment with different values according to server loads. It is not necessarily true that e.g. having a low soft limit, decreasing concurrency and increasing sleep will help handling heavy loads, as it will also increase the backlog of requests the server has to manage in memory. It will also give a much worse experience for genuine connections.

BANDWIDTH_UNIT_COST
The number of bytes, sent and received, by a session that is deemed to cost `1.0`.

The default value `5,000` bytes, meaning the bandwidth cost assigned to a response of 100KB is 20. If your bandwidth is cheap you should probably raise this.

REQUEST_TIMEOUT
An integer number of seconds defaulting to `30`. If a request takes longer than this to respond to, either because of request limiting or because the request is expensive, the server rejects it and returns a timeout error to the client indicating that the server is busy.

This can help prevent large backlogs of unprocessed requests building up under heavy load.

SESSION_TIMEOUT
An integer number of seconds defaulting to `600`. Sessions that have not sent a request for longer than this are disconnected. Properly functioning clients should send a *`server.ping()`* request once roughly 450 seconds have passed since the previous request, in order to avoid disconnection.

4.6.6 Peer Discovery

In response to the *`server.peers.subscribe()`* RPC call, ElectrumX will only return peer servers that it has recently connected to and verified basic functionality.

If you are not running a Tor proxy ElectrumX will be unable to connect to onion server peers, in which case rather than returning no onion peers it will fall back to a hard-coded list.

To give incoming clients a full range of onion servers you will need to be running a Tor proxy for ElectrumX to use.

ElectrumX will perform peer-discovery by default and announce itself to other peers. If your server is private you may wish to disable some of this.

PEER_DISCOVERY
This environment variable is case-insensitive and defaults to `on`.

If `on`, ElectrumX will occasionally connect to and verify its network of peer servers.

If `off`, peer discovery is disabled and a hard-coded default list of servers will be read in and served. If set to `self` then peer discovery is disabled and the server will only return itself in the peers list.

PEER_ANNOUNCE
Set this environment variable to empty to disable announcing itself. If not defined, or non-empty, ElectrumX will announce itself to peers.

If peer discovery is disabled this environment variable has no effect, because ElectrumX only announces itself to peers when doing peer discovery if it notices it is not present in the peer's returned list.

FORCE_PROXY
By default peer discovery happens over the clear internet. Set this to non-empty to force peer discovery to be done via the proxy. This might be useful if you are running a Tor service exclusively and wish to keep your IP address private.

TOR_PROXY_HOST
The host where your Tor proxy is running. Defaults to `localhost`.

If you are not running a Tor proxy just leave this environment variable undefined.

TOR_PROXY_PORT
The port on which the Tor proxy is running. If not set, ElectrumX will autodetect any proxy running on the usual ports 9050 (Tor), 9150 (Tor browser bundle) and 1080 (socks).

4.6.7 Cache

If synchronizing from the Genesis block your performance might change by tweaking the cache size. Cache size is only checked roughly every minute, so the cache can grow beyond the specified size. Moreover, the Python process

is often quite a bit fatter than the cache size, because of Python overhead and also because leveldb consumes a lot of memory when flushing. So I recommend you do not set this over 60% of your available physical RAM:

CACHE_MB
: The amount of cache, in MB, to use. The default is 1,200.

  A portion of the cache is reserved for unflushed history, which is written out frequently. The bulk is used to cache UTXOs.

  Larger caches probably increase performance a little as there is significant searching of the UTXO cache during indexing. However, I don't see much benefit in my tests pushing this too high, and in fact performance begins to fall, probably because LevelDB already caches, and also because of Python GC.

  I do not recommend raising this above 2000.

4.7 Electrum Protocol

This is intended to be a reference for client and server authors alike.

4.7.1 Protocol Basics

Message Stream

Clients and servers communicate using **JSON RPC** over an unspecified underlying stream transport. Examples include TCP, SSL, WS and WSS.

Two standards JSON RPC 1.0 and JSON RPC 2.0 are specified; use of version 2.0 is encouraged but not required. Server support of batch requests is encouraged for version 1.0 but not required.

Note: A client or server should only indicate JSON RPC 2.0 by setting the jsonrpc member of its messages to `"2.0"` if it supports the version 2.0 protocol in its entirety. ElectrumX does and will expect clients advertizing so to function correctly. Those that do not will be disconnected and possibly blacklisted.

Clients making batch requests should limit their size depending on the nature of their query, because servers will limit response size as an anti-DoS mechanism.

Over TCP and SSL raw sockets each RPC call, and each response, MUST be terminated by a single newline to delimit messages. Websocket messages are already framed so they MUST NOT be newline terminated. The JSON specification does not permit control characters within strings, so no confusion is possible there. However it does permit newlines as extraneous whitespace between elements; client and server MUST NOT use newlines in such a way.

If using JSON RPC 2.0's feature of parameter passing by name, the names shown in the description of the method or notification in question MUST be used.

A server advertising support for a particular protocol version MUST support each method documented for that protocol version, unless the method is explicitly marked optional. It may support other methods or additional parameters with unspecified behaviour. Use of additional parameters is discouraged as it may conflict with future versions of the protocol.

Notifications

Some RPC calls are subscriptions which, after the initial response, will send a JSON RPC *notification* each time the thing subscribed to changes. The *method* of the notification is the same as the method of the subscription, and the *params* of the notification (and their names) are given in the documentation of the method.

Version Negotiation

It is desirable to have a way to enhance and improve the protocol without forcing servers and clients to upgrade at the same time.

Protocol versions are denoted by dotted number strings with at least one dot. Examples: "1.5", "1.4.1", "2.0". In "a.b.c" *a* is the major version number, *b* the minor version number, and *c* the revision number.

A party to a connection will speak all protocol versions in a range, say from *protocol_min* to *protocol_max*, which may be the same. When a connection is made, both client and server must initially assume the protocol to use is their own *protocol_min*.

The client should send a *server.version()* RPC call as early as possible in order to negotiate the precise protocol version; see its description for more detail. All responses received in the stream from and including the server's response to this call will use its negotiated protocol version.

Script Hashes

A *script hash* is the hash of the binary bytes of the locking script (ScriptPubKey), expressed as a hexadecimal string. The hash function to use is given by the "hash_function" member of *server.features()* (currently `sha256()` only). Like for block and transaction hashes, when converting the big-endian binary hash to a hexadecimal string the least-significant byte appears first, and the most-significant byte last.

For example, the legacy Bitcoin address from the genesis block:

```
1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa
```

has P2PKH script:

```
76a91462e907b15cbf27d5425399ebf6f0fb50ebb88f1888ac
```

with SHA256 hash:

```
6191c3b590bfcfa0475e877c302da1e323497acf3b42c08d8fa28e364edf018b
```

which is sent to the server reversed as:

```
8b01df4e368ea28f8dc0423bcf7a4923e3a12d307c875e47a0cfbf90b5c39161
```

By subscribing to this hash you can find P2PKH payments to that address.

One public key, the genesis block public key, among the trillions for that address is:

```
04678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e0ea1f61deb
649f6bc3f4cef38c4f35504e51ec112de5c384df7ba0b8d578a4c702b6bf11d5f
```

which has P2PK script:

```
4104678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e0ea1f61deb
649f6bc3f4cef38c4f35504e51ec112de5c384df7ba0b8d578a4c702b6bf11d5fac
```

with SHA256 hash:

```
3318537dfb3135df9f3d950dbdf8a7ae68dd7c7dfef61ed17963ff80f3850474
```

which is sent to the server reversed as:

```
740485f380ff6379d11ef6fe7d7cdd68aea7f8bd0d953d9fdf3531fb7d531833
```

By subscribing to this hash you can find P2PK payments to the genesis block public key.

Note: The Genesis block coinbase is uniquely unspendable and therefore not indexed. It will not show with the above P2PK script hash subscription.

Status

To calculate the *status* of a *script hash* (or address):

1. order confirmed transactions to the script hash by increasing height (and position in the block if there are more than one in a block)

2. form a string that is the concatenation of strings `"tx_hash:height:"` for each transaction in order, where:

   - `tx_hash` is the transaction hash in hexadecimal
   - `height` is the height of the block it is in.

3. Next, with mempool transactions in any order, append a similar string for those transactions, but where **height** is `-1` if the transaction has at least one unconfirmed input, and `0` if all inputs are confirmed.

4. The *status* of the script hash is the `sha256()` hash of the full string expressed as a hexadecimal string, or `null` if the string is empty because there are no transactions.

Block Headers

Originally Electrum clients would download all block headers and verify the chain of hashes and header difficulty in order to confirm the merkle roots with which to check transaction inclusion.

With the Bitcoin chain now past height 500,000, the headers form over 40MB of raw data which becomes 80MB if downloaded as text from Electrum servers. The situation is worse for testnet and coins with more frequent blocks. Downloading and verifying all this data on initial use would take several minutes, during which Electrum was non-responsive.

To facilitate a better experience for SPV clients, particularly on mobile, protocol *version 1.4* introduces an optional *cp_height* argument to the *`blockchain.block.header()`* and *`blockchain.block.headers()`* RPC calls.

This requests the server provide a merkle proof, to a single 32-byte checkpoint hard-coded in the client, that the header(s) provided are valid in the same way the server proves a transaction is included in a block. If several consecutive headers are requested, the proof is provided for the final header - the *prev_hash* links in the headers are sufficient to prove the others valid.

Using this feature client software only needs to download the headers it is interested in up to the checkpoint. Headers after the checkpoint must all be downloaded and validated as before. The RPC calls return the merkle root, so to embed a checkpoint in a client simply make an RPC request to a couple of trusted servers for the greatest height to which a reorganisation of the chain is infeasible, and confirm the returned roots match.

Note: with 500,000 headers of 80 bytes each, a naïve server implementation would require hashing approximately 88MB of data to provide a single merkle proof. ElectrumX implements an optimization such that it hashes only approximately 180KB of data per proof.

4.7.2 Protocol Methods

blockchain.block.header

Return the block header at the given height.

Signature

`blockchain.block.header` (*height*, *cp_height=0*)

New in version 1.3.

Changed in version 1.4: *cp_height* parameter added

Changed in version 1.4.1.

height

The height of the block, a non-negative integer.

cp_height

Checkpoint height, a non-negative integer. Ignored if zero, otherwise the following must hold:

height <= *cp_height*

Result

If *cp_height* is zero, the raw block header as a hexadecimal string.

Otherwise a dictionary with the following keys. This provides a proof that the given header is present in the blockchain; presumably the client has the merkle root hard-coded as a checkpoint.

- *branch*

  The merkle branch of *header* up to *root*, deepest pairing first.

- *header*

  The raw block header as a hexadecimal string. Starting with version 1.4.1, AuxPoW data (if present in the original header) is truncated.

- *root*

  The merkle root of all blockchain headers up to and including *cp_height*.

Example Result

With *height* 5 and *cp_height* 0 on the Bitcoin Cash chain:

```
↪"0100000085144a84488ea88d221c8bd6c059da090e88f8a2c99690ee55dbba4e00000000e11c48fecdd9e72510ca84f023
↪"
```

With *cp_height* 8:

```
{
  "branch": [
     "000000004ebadb55ee9096c9a2f8880e09da59c0d68b1c228da88e48844a1485",
     "96cbbc84783888e4cc971ae8acf86dd3c1a419370336bb3c634c97695a8c5ac9",
     "965ac94082cebbcffe458075651e9cc33ce703ab0115c72d9e8b1a9906b2b636",
     "89e5daa6950b895190716dd26054432b564ccdc2868188ba1da76de8e1dc7591"
     ],
  "header":
↪"0100000085144a84488ea88d221c8bd6c059da090e88f8a2c99690ee55dbba4e00000000e11c48fecdd9e72510ca84f023
↪",
  "root": "e347b1c43fd9b5415bf0d92708db8284b78daf4d0e24f9c3405f45feb85e25db"
}
```


blockchain.block.headers

Return a concatenated chunk of block headers from the main chain.

Signature

`blockchain.block.`**`headers`**(*start_height*, *count*, *cp_height=0*)

New in version 1.2.

Changed in version 1.4: *cp_height* parameter added

Changed in version 1.4.1.

start_height

The height of the first header requested, a non-negative integer.

count

The number of headers requested, a non-negative integer.

cp_height

Checkpoint height, a non-negative integer. Ignored if zero, otherwise the following must hold:

start_height + (*count* - 1) <= *cp_height*

Result

A dictionary with the following members:

- *count*

  The number of headers returned, between zero and the number requested. If the chain has not extended sufficiently far, only the available headers will be returned. If more headers than *max* were requested at most *max* will be returned.

- *hex*

  The binary block headers concatenated together in-order as a hexadecimal string. Starting with version 1.4.1, AuxPoW data (if present in the original header) is truncated if *cp_height* is nonzero.

- *max*

  The maximum number of headers the server will return in a single request.

The dictionary additionally has the following keys if *count* and *cp_height* are not zero. This provides a proof that all the given headers are present in the blockchain; presumably the client has the merkle root hard-coded as a checkpoint.

- *root*

  The merkle root of all blockchain headers up to and including *cp_height*.

- *branch*

  The merkle branch of the last returned header up to *root*, deepest pairing first.

Example Response

See *here* for an example of *root* and *branch* keys.

```
{
  "count": 2,
  "hex":
↪"01000000000000000000000000000000000000000000000000000000000000000000003ba3edfd7a7b12b27ac72c3e67
↪"
  "max": 2016
}
```

blockchain.estimatefee

Return the estimated transaction fee per kilobyte for a transaction to be confirmed within a certain number of blocks.

Signature

`blockchain.estimatefee(number)`

Deprecated since version 1.4.2.

number

The number of blocks to target for confirmation.

Result

The estimated transaction fee in coin units per kilobyte, as a floating point number. If the daemon does not have enough information to make an estimate, the integer `-1` is returned.

Example Result

```
0.00001
```

blockchain.headers.subscribe

Subscribe to receive block headers when a new block is found.

Signature

`blockchain.headers.subscribe()`

Result

The header of the current block chain tip. The result is a dictionary with two members:

- *hex*

  The binary header as a hexadecimal string.

- *height*

  The height of the header, an integer.

Example Result

```
{
  "height": 520481,
  "hex":
↪"00000020890208a0ae3a3892aa047c5468725846577cfcd9b512b50000000000000000005dc2b02f2d297a9064ee10303
↪"
}
```


Notifications

As this is a subscription, the client will receive a notification when a new block is found. The notification's signature is:

`blockchain.headers.`**`subscribe`**`(`*header*`)`

- *header*

  See **Result** above.

Note: should a new block arrive quickly, perhaps while the server is still processing prior blocks, the server may only notify of the most recent chain tip. The protocol does not guarantee notification of all intermediate block headers.

In a similar way the client must be prepared to handle chain reorganisations. Should a re-org happen the new chain tip will not sit directly on top of the prior chain tip. The client must be able to figure out the common ancestor block and request any missing block headers to acquire a consistent view of the chain state.

---

blockchain.relayfee

Return the minimum fee a low-priority transaction must pay in order to be accepted to the daemon's memory pool.

Signature

`blockchain.`**`relayfee`**`()`

Deprecated since version 1.4.2.

Result

The fee in whole coin units as a floating point number.

Example Results


```
0.000001
```


blockchain.scripthash.get_balance

Return the confirmed and unconfirmed balances of a *script hash*.

Signature

`blockchain.scripthash.`**`get_balance`**`(`*scripthash*`)`

New in version 1.1.

scripthash

The script hash as a hexadecimal string.

Result

A dictionary with keys *confirmed* and *unconfirmed*. The value of each is the appropriate balance in coin units as a string.

Result Example

```
{
  "confirmed": "1.03873966",
  "unconfirmed": "0.236844"
}
```

blockchain.scripthash.get_history

Return the confirmed and unconfirmed history of a *script hash*.

Signature

`blockchain.scripthash.`**`get_history`**`(`*scripthash*`)`

New in version 1.1.

scripthash

The script hash as a hexadecimal string.

Result

A list of confirmed transactions in blockchain order, with the output of *`blockchain.scripthash.get_mempool()`* appended to the list. Each confirmed transaction is a dictionary with the following keys:

- *height*

  The integer height of the block the transaction was confirmed in.

- *tx_hash*

  The transaction hash in hexadecimal.

See *`blockchain.scripthash.get_mempool()`* for how mempool transactions are returned.

Result Examples

```
[
  {
    "height": 200004,
    "tx_hash": "acc3758bd2a26f869fcc67d48ff30b96464d476bca82c1cd6656e7d506816412"
  },
  {
    "height": 215008,
    "tx_hash": "f3e1bf48975b8d6060a9de8884296abb80be618dc00ae3cb2f6cee3085e09403"
  }
]
```

```
[
  {
    "fee": 20000,
    "height": 0,
    "tx_hash": "9fbed79a1e970343fcd39f4a2d830a6bde6de0754ed2da70f489d0303ed558ec"
  }
]
```

blockchain.scripthash.get_mempool

Return the unconfirmed transactions of a *script hash*.

Signature

`blockchain.scripthash.`**`get_mempool`**`(`*scripthash*`)`

New in version 1.1.

scripthash

The script hash as a hexadecimal string.

Result

A list of mempool transactions in arbitrary order. Each mempool transaction is a dictionary with the following keys:

- *height*

  `0` if all inputs are confirmed, and `-1` otherwise.

- *tx_hash*

  The transaction hash in hexadecimal.

- *fee*

  The transaction fee in minimum coin units (satoshis).

Result Example

```
[
  {
    "tx_hash": "45381031132c57b2ff1cbe8d8d3920cf9ed25efd9a0beb764bdb2f24c7d1c7e3",
    "height": 0,
    "fee": 24310
  }
]
```

blockchain.scripthash.listunspent

Return an ordered list of UTXOs sent to a script hash.

Signature

`blockchain.scripthash.`**`listunspent`**`(`*scripthash*`)`

New in version 1.1.

scripthash

The script hash as a hexadecimal string.

Result

A list of unspent outputs in blockchain order. This function takes the mempool into account. Mempool transactions paying to the address are included at the end of the list in an undefined order. Any output that is spent in the mempool does not appear. Each output is a dictionary with the following keys:

- *height*

  The integer height of the block the transaction was confirmed in. `0` if the transaction is in the mempool.

- *tx_pos*

  The zero-based index of the output in the transaction's list of outputs.

- *tx_hash*

  The output's transaction hash as a hexadecimal string.

- *value*

  The output's value in minimum coin units (satoshis).

Result Example

```
[
  {
    "tx_pos": 0,
    "value": 45318048,
    "tx_hash": "9f2c45a12db0144909b5db269415f7319179105982ac70ed80d76ea79d923ebf",
    "height": 437146
  },
  {
    "tx_pos": 0,
    "value": 919195,
    "tx_hash": "3d2290c93436a3e964cfc2f0950174d8847b1fbe3946432c4784e168da0f019f",
    "height": 441696
  }
]
```

blockchain.scripthash.subscribe

Subscribe to a script hash.

Signature

`blockchain.scripthash.`**`subscribe`**(*scripthash*)

New in version 1.1.

scripthash

The script hash as a hexadecimal string.

Result

The *status* of the script hash.

Notifications

The client will receive a notification when the *status* of the script hash changes. Its signature is

`blockchain.scripthash.`**`subscribe`**(*scripthash*, *status*)

blockchain.scripthash.unsubscribe

Unsubscribe from a script hash, preventing future notifications if its *status* changes.

Signature

`blockchain.scripthash.`**`unsubscribe`**(*scripthash*)

New in version 1.4.2.

scripthash

The script hash as a hexadecimal string.

Result

Returns `True` if the scripthash was subscribed to, otherwise `False`. Note that `False` might be returned even for something subscribed to earlier, because the server can drop subscriptions in rare circumstances.

blockchain.transaction.broadcast

Broadcast a transaction to the network.

Signature

`blockchain.transaction.`**`broadcast`**(*raw_tx*)

Changed in version 1.1: errors returned as JSON RPC errors rather than as a result.

raw_tx

The raw transaction as a hexadecimal string.

Result

The transaction hash as a hexadecimal string.

Note protocol version 1.0 (only) does not respond according to the JSON RPC specification if an error occurs. If the daemon rejects the transaction, the result is the error message string from the daemon, as if the call were successful. The client needs to determine if an error occurred by comparing the result to the expected transaction hash.

Result Examples

```
"a76242fce5753b4212f903ff33ac6fe66f2780f34bdb4b33b175a7815a11a98e"
```

Protocol version 1.0 returning an error as the result:

```
"258: txn-mempool-conflict"
```

blockchain.transaction.get

Return a raw transaction.

Signature

`blockchain.transaction.`**`get`**(*tx_hash*, *verbose=false*)

Changed in version 1.1: ignored argument *height* removed

Changed in version 1.2: *verbose* argument added

tx_hash

The transaction hash as a hexadecimal string.

verbose

Whether a verbose coin-specific response is required.

Result

If *verbose* is `false`:

The raw transaction as a hexadecimal string.

If *verbose* is `true`:

The result is a coin-specific dictionary – whatever the coin daemon returns when asked for a verbose form of the raw transaction.

Example Results

When *verbose* is `false`:

```
"01000000015bb9142c960a838329694d3fe9ba08c2a6421c5158d8f7044cb7c48006c1b48"
"4000000006a4730440220229ea5359a63c2b83a713fcc20d8c41b20d48fe639a639d2a824"
"6a137f29d0fc02201de12de9c056912a4e581a62d12fb5f43ee6c08ed0238c32a1ee76921"
"3ca8b8b412103bcf9a004f1f7a9a8d8acce7b51c983233d107329ff7c4fb53e44c855dbe1"
"f6a4feffffff02c6b68200000000001976a9141041fb024bd7a1338ef1959026bbba86006"
"4fe5f88ac50a8cf00000000001976a91445dac110239a7a3814535c15858b939211f85298"
"88ac61ee0700"
```

When *verbose* is `true`:

```
{
  "blockhash": "000000000000000015a4f37ece911e5e3549f988e855548ce7494a0a08b2ad6",
  "blocktime": 1520074861,
  "confirmations": 679,
  "hash": "36a3692a41a8ac60b73f7f41ee23f5c917413e5b2fad9e44b34865bd0d601a3d",
  "hex":
↪"01000000015bb9142c960a838329694d3fe9ba08c2a6421c5158d8f7044cb7c48006c1b484000000006a4730440220229e
↪",
  "locktime": 519777,
  "size": 225,
  "time": 1520074861,
  "txid": "36a3692a41a8ac60b73f7f41ee23f5c917413e5b2fad9e44b34865bd0d601a3d",
  "version": 1,
  "vin": [ {
    "scriptSig": {
      "asm":
↪"30440220229ea5359a63c2b83a713fcc20d8c41b20d48fe639a639d2a8246a137f29d0fc02201de12de9c056912a4e581a
↪03bcf9a004f1f7a9a8d8acce7b51c983233d107329ff7c4fb53e44c855dbe1f6a4",
      "hex":
↪"4730440220229ea5359a63c2b83a713fcc20d8c41b20d48fe639a639d2a8246a137f29d0fc02201de12de9c056912a4e58
↪"
    },
    "sequence": 4294967294,
    "txid": "84b4c10680c4b74c04f7d858511c42a6c208bae93f4d692983830a962c14b95b",
    "vout": 0}],
  "vout": [ { "n": 0,
              "scriptPubKey": { "addresses": [ "12UxrUZ6tyTLoR1rT1N4nuCgS9DDURTJgP"],
                                "asm": "OP_DUP OP_HASH160␣
↪1041fb024bd7a1338ef1959026bbba860064fe5f OP_EQUALVERIFY OP_CHECKSIG",
                                "hex":
↪"76a9141041fb024bd7a1338ef1959026bbba860064fe5f88ac",
                                "reqSigs": 1,
                                "type": "pubkeyhash"},
              "value": 0.0856647},
            { "n": 1,
              "scriptPubKey": { "addresses": [ "17NMgYPrguizvpJmB1Sz62ZHeeFydBYbZJ"],
                                "asm": "OP_DUP OP_HASH160␣
↪45dac110239a7a3814535c15858b939211f85298 OP_EQUALVERIFY OP_CHECKSIG",
```

(continues on next page)

(continued from previous page)

```
                                "hex":
↪"76a91445dac110239a7a3814535c15858b939211f8529888ac",
                                "reqSigs": 1,
                                "type": "pubkeyhash"},
             "value": 0.1360904}]}
```

blockchain.transaction.get_merkle

Return the merkle branch to a confirmed transaction given its hash and height.

Signature

`blockchain.transaction.`**`get_merkle`**(*tx_hash*, *height*)

tx_hash

The transaction hash as a hexadecimal string.

height

The height at which it was confirmed, an integer.

Result

A dictionary with the following keys:

- *block_height*

  The height of the block the transaction was confirmed in.

- *merkle*

  A list of transaction hashes the current hash is paired with, recursively, in order to trace up to obtain merkle root of the block, deepest pairing first.

- *pos*

  The 0-based index of the position of the transaction in the ordered list of transactions in the block.

Result Example

```
{
  "merkle":
  [
    "713d6c7e6ce7bbea708d61162231eaa8ecb31c4c5dd84f81c20409a90069cb24",
    "03dbaec78d4a52fbaf3c7aa5d3fccd9d8654f323940716ddf5ee2e4bda458fde",
    "e670224b23f156c27993ac3071940c0ff865b812e21e0a162fe7a005d6e57851",
    "369a1619a67c3108a8850118602e3669455c70cdcdb89248b64cc6325575b885",
    "4756688678644dcb27d62931f04013254a62aeee5dec139d1aac9f7b1f318112",
    "7b97e73abc043836fd890555bfce54757d387943a6860e5450525e8e9ab46be5",
    "61505055e8b639b7c64fd58bce6fc5c2378b92e025a02583303f69930091b1c3",
    "27a654ff1895385ac14a574a0415d3bbba9ec23a8774f22ec20d53dd0b5386ff",
    "5312ed87933075e60a9511857d23d460a085f3b6e9e5e565ad2443d223cfccdc",
    "94f60b14a9f106440a197054936e6fb92abbd69d6059b38fdf79b33fc864fca0",
    "2d64851151550e8c4d337f335ee28874401d55b358a66f1bafab2c3e9f48773d"
  ],
  "block_height": 450538,
  "pos": 710
}
```

blockchain.transaction.id_from_pos

Return a transaction hash and optionally a merkle proof, given a block height and a position in the block.

Signature

`blockchain.transaction.`**`id_from_pos`**(*height*, *tx_pos*, *merkle=false*)

New in version 1.4.

height

The main chain block height, a non-negative integer.

tx_pos

A zero-based index of the transaction in the given block, an integer.

merkle

Whether a merkle proof should also be returned, a boolean.

Result

If *merkle* is `false`, the transaction hash as a hexadecimal string. If `true`, a dictionary with the following keys:

- *tx_hash*

  The transaction hash as a hexadecimal string.

- *merkle*

  A list of transaction hashes the current hash is paired with, recursively, in order to trace up to obtain merkle root of the block, deepest pairing first.

Example Results

When *merkle* is `false`:

```
"fc12dfcb4723715a456c6984e298e00c479706067da81be969e8085544b0ba08"
```

When *merkle* is `true`:

```
{
  "tx_hash": "fc12dfcb4723715a456c6984e298e00c479706067da81be969e8085544b0ba08",
  "merkle":
  [
    "928c4275dfd6270349e76aa5a49b355eefeb9e31ffbe95dd75fed81d219a23f8",
    "5f35bfb3d5ef2ba19e105dcd976928e675945b9b82d98a93d71cbad0e714d04e",
    "f136bcffeeed8844d54f90fc3ce79ce827cd8f019cf1d18470f72e4680f99207",
    "6539b8ab33cedf98c31d4e5addfe40995ff96c4ea5257620dfbf86b34ce005ab",
    "7ecc598708186b0b5bd10404f5aeb8a1a35fd91d1febbb2aac2d018954885b1e",
    "a263aae6c470b9cde03b90675998ff6116f3132163911fafbeeb7843095d3b41",
    "c203983baffe527edb4da836bc46e3607b9a36fa2c6cb60c1027f0964d971b29",
    "306d89790df94c4632d652d142207f53746729a7809caa1c294b895a76ce34a9",
    "c0b4eff21eea5e7974fe93c62b5aab51ed8f8d3adad4583c7a84a98f9e428f04",
    "f0bd9d2d4c4cf00a1dd7ab3b48bbbb4218477313591284dcc2d7ca0aaa444e8d",
    "503d3349648b985c1b571f59059e4da55a57b0163b08cc50379d73be80c4c8f3"
  ]
}
```

mempool.get_fee_histogram

Return a histogram of the fee rates paid by transactions in the memory pool, weighted by transaction size.

Signature

`mempool.`**`get_fee_histogram`**`()`

New in version 1.2.

Deprecated since version 1.4.2.

Result

The histogram is an array of [*fee*, *vsize*] pairs, where $vsize_n$ is the cumulative virtual size of mempool transactions with a fee rate in the interval [fee_{n-1}, fee_n], and $fee_{n-1} > fee_n$.

Fee intervals may have variable size. The choice of appropriate intervals is currently not part of the protocol.

Example Result

```
[[12, 128812], [4, 92524], [2, 6478638], [1, 22890421]]
```

server.add_peer

A newly-started server uses this call to get itself into other servers' peers lists. It should not be used by wallet clients.

Signature

`server.`**`add_peer`**`(`*features*`)`

New in version 1.1.

- *features*

  The same information that a call to the sender's *server.features()* RPC call would return.

Result

A boolean indicating whether the request was tentatively accepted. The requesting server will appear in *server.peers.subscribe()* when further sanity checks complete successfully.

server.banner

Return a banner to be shown in the Electrum console.

Signature

`server.`**`banner`**`()`

Result

A string.

Example Result

```
"Welcome to Electrum!"
```

server.donation_address

Return a server donation address.

Signature

`server.donation_address()`

Result

A string.

Example Result

```
"1BWwXJH3q6PRsizBkSGm2Uw4Sz1urZ5sCj"
```

server.features

Return a list of features and services supported by the server.

Signature

`server.features()`

Result

A dictionary of keys and values. Each key represents a feature or service of the server, and the value gives additional information.

The following features MUST be reported by the server. Additional key-value pairs may be returned.

- *hosts*

  A dictionary, keyed by host name, that this server can be reached at. Normally this will only have a single entry; other entries can be used in case there are other connection routes (e.g. Tor).

  The value for a host is itself a dictionary, with the following optional keys:

  - *ssl_port*

    An integer. Omit or set to `null` if SSL connectivity is not provided.

  - *tcp_port*

    An integer. Omit or set to `null` if TCP connectivity is not provided.

  A server should ignore information provided about any host other than the one it connected to.

- *genesis_hash*

  The hash of the genesis block. This is used to detect if a peer is connected to one serving a different network.

- *hash_function*

  The hash function the server uses for *script hashing*. The client must use this function to hash pay-to-scripts to produce script hashes to send to the server. The default is "sha256". "sha256" is currently the only acceptable value.

- *server_version*

  A string that identifies the server software. Should be the same as the first element of the result to the `server.version()` RPC call.

- *protocol_max*

- *protocol_min*

  Strings that are the minimum and maximum Electrum protocol versions this server speaks. Example: "1.1".

- *pruning*

  An integer, the pruning limit. Omit or set to `null` if there is no pruning limit. Should be the same as what would suffix the letter `p` in the IRC real name.

Example Result

```
{
    "genesis_hash": "000000000933ea01ad0ee984209779baaec3ced90fa3f408719526f8d77f4943
↪",
    "hosts": {"14.3.140.101": {"tcp_port": 51001, "ssl_port": 51002}},
    "protocol_max": "1.0",
    "protocol_min": "1.0",
    "pruning": null,
    "server_version": "ElectrumX 1.0.17",
    "hash_function": "sha256"
}
```

server.peers.subscribe

Return a list of peer servers. Despite the name this is not a subscription and the server must send no notifications.

Signature

`server.peers.`**`subscribe`**`()`

Result

An array of peer servers, each returned as a 3-element array. For example:

```
["107.150.45.210",
 "e.anonyhost.org",
 ["v1.0", "p10000", "t", "s995"]]
```

The first element is the IP address, the second is the host name (which might also be an IP address), and the third is a list of server features. Each feature and starts with a letter. 'v' indicates the server maximum protocol version, 'p' its pruning limit and is omitted if it does not prune, 't' is the TCP port number, and 's' is the SSL port number. If a port is not given for 's' or 't' the default port for the coin network is implied. If 's' or 't' is missing then the server does not support that transport.

server.ping

Ping the server to ensure it is responding, and to keep the session alive. The server may disconnect clients that have sent no requests for roughly 10 minutes.

Signature

`server.`**`ping`**`()`

New in version 1.2.

Result

Returns `null`.

server.version

Identify the client to the server and negotiate the protocol version. Only the first *server.version()* message is accepted.

Signature

server.**version**(*client_name=""*, *protocol_version="1.4"*)

- *client_name*

  A string identifying the connecting client software.

- *protocol_version*

  An array `[protocol_min, protocol_max]`, each of which is a string. If `protocol_min` and `protocol_max` are the same, they can be passed as a single string rather than as an array of two strings, as for the default value.

The server should use the highest protocol version both support:

```
version = min(client.protocol_max, server.protocol_max)
```

If this is below the value:

```
max(client.protocol_min, server.protocol_min)
```

then there is no protocol version in common and the server must close the connection. Otherwise it should send a response appropriate for that protocol version.

Result

An array of 2 strings:

```
[server_software_version, protocol_version]
```

identifying the server and the protocol version that will be used for future communication.

Example:

```
server.version("Electrum 3.0.6", ["1.1", "1.2"])
```

Example Result:

```
["ElectrumX 1.2.1", "1.2"]
```

Masternode methods (Dash and compatible coins)

masternode.announce.broadcast

Pass through the masternode announce message to be broadcast by the daemon.

Whenever a masternode comes online or a client is syncing, they will send this message which describes the masternode entry and how to validate messages from it.

Signature

masternode.announce.**broadcast**(*signmnb*)

- *signmnb*

  Signed masternode broadcast message in hexadecimal format.

Result

`true` if the message was broadcasted successfully otherwise `false`.

Example:

```
masternode.announce.broadcast(
↪"012b825a65a24e2eb8edadbe27c4716dab993bf1046a66da77268ec87dbdd9dfc80100000000ffffffff0000000000000
↪")
```

Example Result:

```
true
```

masternode.subscribe

Returns the status of masternode.

Signature

`masternode.`**`subscribe`**(*collateral*)

- *collateral*

  The txId and the index of the collateral.

  A masternode collateral is a transaction with a specific amount of coins, it's also known as a masternode identifier.

  i.e. for DASH the required amount is 1,000 DASH or for $PAC is 500,000 $PAC.

Result

As this is a subscription, the client will receive a notification when the masternode status changes.

The status depends on the server the masternode is hosted, the internet connection, the offline time and even the collateral amount, so this subscription notice these changes to the user.

Example:

```
masternode.subscribe(
↪"8c59133e714797650cf69043d05e409bbf45670eed7c4e4a386e52c46f1b5e24-0")
```

Example Result:

```
{'method': 'masternode.subscribe', u'jsonrpc': u'2.0', u'result': u'ENABLED', 'params
↪': ['8c59133e714797650cf69043d05e409bbf45670eed7c4e4a386e52c46f1b5e24-0'], u'id':␣
↪19}
```

masternode.list

Returns the list of masternodes.

Signature

`masternode.`**`list`**(*payees*)

- *payees*

  An array of masternode payee addresses.

Result

An array with the masternodes information.

Example:

```
masternode.list("['PDFHmjKLvSGdnWgDJSJX49Rrh0SJtRANcE',
'PDFHmjKLvSGdnWgDJSJX49Rrh0SJtRANcF']")
```

Example Result:

```
[
  {
    "vin": "9d298c00dae8b491d6801f50cab2e0037852cb556c5619ddb07c50421x9a31ab",
    "status": "ENABLED",
    "protocol": 70213,
    "payee": "PDFHmjKLvSGdnWgDJSJX49Rrh0SJtRANcE",
    "lastseen": "2018-04-01 12:34",
    "activeseconds": 1258000,
    "lastpaidtime": "2018-03-10 12:29",
    "lastpaidblock": 1234,
    "ip": "1.0.0.1",
    "paymentposition": 184,
    "inselection": true,
    "balance": 510350
  },
  {
    "vin": "9d298c00dae8b491d6801f50cab2e0037852cb556c5619ddb07c50421x9a31ac",
    "status": "ENABLED",
    "protocol": 70213,
    "payee": "PDFHmjKLvSGdnWgDJSJX49Rrh0SJtRANcF",
    "lastseen": "2018-04-01 12:34",
    "activeseconds": 1258000,
    "lastpaidtime": "2018-03-15 05:29",
    "lastpaidblock": 1234,
    "ip": "1.0.0.2",
    "paymentposition": 3333,
    "inselection": false,
    "balance": 520700
  },
  ...,
  ...,
  ...,
  ...
]
```

ProTx methods (Dash DIP3)

protx.diff

Returns a diff between two deterministic masternode lists. The result also contains proof data.

Signature

protx.**diff** (*base_height*, *height*)

base_height

The starting block height

1 <= base_height

height

The ending block height.

base_height <= *height*

Result

A dictionary with deterministic masternode lists diff plus proof data

Example:

```
protx.diff(1, 20000)
```

Example Result:


```
{
  "baseBlockHash": "000000000b866e7fefc7df2b4b37f236175cee9ab6dc925a30c62401d92b7406",
  "blockHash": "0000000005b3f97e0af8c72f9a96eca720237e374ca860938ba0d7a68471c4d6",
  "cbTxMerkleTree":
↪"0200000002c9802d02435cfe09e4253bc1ba4875e9a2f920d5d6adf005d5b9306e5322e6f476d885273422c2fe18e8c420
↪",
  "cbTx":
↪"0300050001000000000000000000000000000000000000000000000000000000000000ffffffff4b02204e0478673
↪",
  "deletedMNs": [
  ],
  "mnList": [
    {
      "proRegTxHash":
↪"6f0bdd7034ce8d3a6976a15e4b4442c274b5c1739fb63fc0a50f01425580e17e",
      "confirmedHash":
↪"000000000be653cd1fbc213239cfec83ca68da657f24cc05305d0be75d34e392",
      "service": "173.61.30.231:19023",
      "pubKeyOperator":
↪"8da7ee1a40750868badef2c17d5385480cae7543f8d4d6e5f3c85b37fdd00a6b4f47726b96e7e7c7a3ea68b5d5cb2196
↪",
      "keyIDVoting": "b35c75cbc69433175d3459843e1f6ebe145bf6a3",
      "isValid": true
    }
  ],
  "merkleRootMNList":
↪"17dc82a12c7eaeb8d7efcd0b64af236b836bede2095dec5b3072408ed8c301b3"
}
```


protx.info

Returns detailed information about a deterministic masternode.

Signature

`protx.info`(*protx_hash*)

protx_hash

The hash of the initial ProRegTx.

Result

A dictionary with detailed deterministic masternode data

Example:

```
protx.info("6f0bdd7034ce8d3a6976a15e4b4442c274b5c1739fb63fc0a50f01425580e17e")
```

Example Result:

```
{
  "proTxHash": "6f0bdd7034ce8d3a6976a15e4b4442c274b5c1739fb63fc0a50f01425580e17e",
  "collateralHash": "b41439376b6117aebe6ad1ce31dcd217d4934fd00c104029ecb7d21c11d17c94
↪",
  "collateralIndex": 3,
  "operatorReward": 0,
  "state": {
    "registeredHeight": 19525,
    "lastPaidHeight": 20436,
    "PoSePenalty": 0,
    "PoSeRevivedHeight": -1,
    "PoSeBanHeight": -1,
    "revocationReason": 0,
    "keyIDOwner": "b35c75cbc69433175d3459843e1f6ebe145bf6a3",
    "pubKeyOperator":
↪"8da7ee1a40750868badef2c17d5385480cae7543f8d4d6e5f3c85b37fdd00a6b4f47726b96e7e7c7a3ea68b5d5cb2196
↪",
    "keyIDVoting": "b35c75cbc69433175d3459843e1f6ebe145bf6a3",
    "ownerKeyAddr": "ybGQ7a6e7dkJY2jxdbDwdBtyjKZJ8VB7YC",
    "votingKeyAddr": "ybGQ7a6e7dkJY2jxdbDwdBtyjKZJ8VB7YC",
    "addr": "173.61.30.231:19023",
    "payoutAddress": "yWdXnYxGbouNoo8yMvcbZmZ3Gdp6BpySxL"
  },
  "confirmations": 984
}
```

4.7.3 Protocol Changes

This documents lists changes made by protocol version.

Version 1.0

Deprecated methods

- *blockchain.utxo.get_address()*
- *blockchain.numblocks.subscribe()*

Version 1.1

Changes

- improved semantics of *server.version()* to aid protocol negotiation, and a changed return value.
- *blockchain.transaction.get()* no longer takes the *height* argument that was ignored anyway.
- *blockchain.transaction.broadcast()* returns errors like any other JSON RPC call. A transaction hash result is only returned on success.

New methods

- *blockchain.scripthash.get_balance()*
- *blockchain.scripthash.get_history()*
- *blockchain.scripthash.get_mempool()*
- *blockchain.scripthash.listunspent()*
- *blockchain.scripthash.subscribe()*
- *server.features()*
- *server.add_peer()*

Removed methods

- *blockchain.utxo.get_address()*
- *blockchain.numblocks.subscribe()*

Version 1.2

Changes

- *blockchain.transaction.get()* now has an optional parameter *verbose*.
- *blockchain.headers.subscribe()* now has an optional parameter *raw*.
- *server.version()* should not be used for "ping" functionality; use the new *server.ping()* method instead.

New methods

- *blockchain.block.headers()*
- *mempool.get_fee_histogram()*
- *server.ping()*

Deprecated methods

- *blockchain.block.get_chunk()*. Switch to *blockchain.block.headers()*
- *blockchain.address.get_balance()*. Switch to *blockchain.scripthash.get_balance()*.
- *blockchain.address.get_history()*. Switch to *blockchain.scripthash.get_history()*.
- *blockchain.address.get_mempool()*. Switch to *blockchain.scripthash.get_mempool()*.
- *blockchain.address.listunspent()*. Switch to *blockchain.scripthash.listunspent()*.
- *blockchain.address.subscribe()*. Switch to *blockchain.scripthash.subscribe()*.

- *blockchain.headers.subscribe()* with *raw* other than `True`.

Version 1.3

Changes

- *blockchain.headers.subscribe()* argument *raw* switches default to `True`

New methods

- *blockchain.block.header()*

Removed methods

- *blockchain.address.get_balance()*
- *blockchain.address.get_history()*
- *blockchain.address.get_mempool()*
- *blockchain.address.listunspent()*
- *blockchain.address.subscribe()*

Deprecated methods

- *blockchain.block.get_header()*. Switch to *blockchain.block.header()*.

Version 1.4

This version removes all support for *deserialized headers*.

Changes

- Deserialized headers are no longer available, so removed argument *raw* from *blockchain.headers.subscribe()*.
- Only the first *server.version()* message is accepted.
- Optional *cp_height* argument added to *blockchain.block.header()* and *blockchain.block.headers()* to return merkle proofs of the header to a given checkpoint.

New methods

- *blockchain.transaction.id_from_pos()* to return a transaction hash, and optionally a merkle proof, given a block height and position in the block.

Removed methods

- *blockchain.block.get_header()*
- *blockchain.block.get_chunk()*

Version 1.4.1

Changes

- *blockchain.block.header()* and *blockchain.block.headers()* now truncate AuxPoW data (if using an AuxPoW chain) when *cp_height* is nonzero. AuxPoW data is still present when *cp_height* is zero. Non-AuxPoW chains are unaffected.

Version 1.4.1

New methods

- `blockchain.scipthash.unsubscribe()` to unsubscribe from a script hash.

4.7.4 Removed Protocol Methods

This documents protocol methods that are still supported in some protocol versions, but not the most recent one.

Deserialized Headers

A *deserialized header* is a dictionary describing a block at a given height.

A typical example would be similar to this template:

```
{
  "block_height": <integer>,
  "version": <integer>,
  "prev_block_hash": <hexadecimal string>,
  "merkle_root":  <hexadecimal string>,
  "timestamp": <integer>,
  "bits": <integer>,
  "nonce": <integer>
}
```

Note: The precise format of a deserialized block header varies by coin, and also potentially by height for the same coin. Detailed knowledge of the meaning of a block header is neither necessary nor appropriate in the server. Consequently they were removed from the protocol in version 1.4.

blockchain.address.get_balance

Return the confirmed and unconfirmed balances of a bitcoin address.

Signature

blockchain.address.**get_balance**(*address*)

Deprecated since version 1.2: removed in version 1.3

- *address*

  The address as a Base58 string.

Result

See *blockchain.scripthash.get_balance()*.

blockchain.address.get_history

Return the confirmed and unconfirmed history of a bitcoin address.

Signature

blockchain.address.**get_history**(*address*)

Deprecated since version 1.2: removed in version 1.3

- *address*

  The address as a Base58 string.

Result

As for *blockchain.scripthash.get_history()*.

blockchain.address.get_mempool

Return the unconfirmed transactions of a bitcoin address.

Signature

blockchain.address.**get_mempool**(*address*)

Deprecated since version 1.2: removed in version 1.3

- *address*

  The address as a Base58 string.

Result

As for *blockchain.scripthash.get_mempool()*.

blockchain.address.listunspent

Return an ordered list of UTXOs sent to a bitcoin address.

Signature

blockchain.address.**listunspent**(*address*)

Deprecated since version 1.2: removed in version 1.3

- *address*

  The address as a Base58 string.

Result

As for *blockchain.scripthash.listunspent()*.

blockchain.address.subscribe

Subscribe to a bitcoin address.

Signature

`blockchain.address.`**`subscribe`**(*address*)

Deprecated since version 1.2: removed in version 1.3

address

The address as a Base58 string.

Result

The *status* of the address.

Notifications

As this is a subscription, the client will receive a notification when the *status* of the address changes. Its signature is

`blockchain.address.`**`subscribe`**(*address*, *status*)

blockchain.headers.subscribe

Subscribe to receive block headers when a new block is found.

Signature

Changed in version 1.2: Optional *raw* parameter added, defaulting to `false`.

Changed in version 1.3: *raw* parameter defaults to `true`.

Changed in version 1.4: *raw* parameter removed; responses and notifications pass raw headers.

- *raw*

  This single boolean argument exists in protocol versions 1.2 (defaulting to `false`) and 1.3 (defaulting to `true`) only.

Result

The header of the current block chain tip. If *raw* is `true` the result is a dictionary with two members:

- *hex*

  The binary header as a hexadecimal string.

- *height*

  The height of the header, an integer.

If *raw* is `false` the result is the coin-specific *deserialized header*.

Example Result

With *raw* `false`:

```
{
  "bits": 402858285,
  "block_height": 520481,
  "merkle_root":
↪"8e8e932eb858fd53cf09943d7efc9a8f674dc1363010ee64907a292d2fb0c25d",
  "nonce": 3288656012,
  "prev_block_hash":
↪"00000000000000000b512b5d9fc7c5746587268547c04aa92383aaea0080289",
  "timestamp": 1520495819,
  "version": 536870912
}
```


With *raw* `true`:


```
{
  "height": 520481,
  "hex":
↪"00000020890208a0ae3a3892aa047c5468725846577cfcd9b512b50000000000000000005dc2b02f2d297a9064ee1
↪"
}
```


Notifications

As this is a subscription, the client will receive a notification when a new block is found. The notification's signature is:

- *header*

  See **Result** above.

Note: should a new block arrive quickly, perhaps while the server is still processing prior blocks, the server may only notify of the most recent chain tip. The protocol does not guarantee notification of all intermediate block headers.

In a similar way the client must be prepared to handle chain reorganisations. Should a re-org happen the new chain tip will not sit directly on top of the prior chain tip. The client must be able to figure out the common ancestor block and request any missing block headers to acquire a consistent view of the chain state.

---

blockchain.numblocks.subscribe

Subscribe to receive the block height when a new block is found.

Signature

`blockchain.numblocks.`**`subscribe`**`()`

Deprecated since version 1.0: removed in version 1.1

Result

The height of the current block, an integer.

Notifications

As this is a subscription, the client will receive a notification when a new block is found. The notification's signature is:

`blockchain.numblocks.`**`subscribe`**`(`*`height`*`)`

blockchain.utxo.get_address

Return the address paid to by a UTXO.

Signature

`blockchain.utxo.`**`get_address`**(*tx_hash*, *index*)
Optional in version 1.0, removed in version 1.1

tx_hash

The transaction hash as a hexadecimal string.

index

The zero-based index of the UTXO in the transaction.

Result

A Base58 address string, or `null`. If the transaction doesn't exist, the index is out of range, or the output is not paid to an address, `null` must be returned. If the output is spent `null` *may* be returned.

blockchain.block.get_header

Return the *deserialized header* of the block at the given height.

Signature

`blockchain.block.`**`get_header`**(*height*)

Deprecated since version 1.3: removed in version 1.4

height

The height of the block, an integer.

Result

The coin-specific *deserialized header*.

Example Result

```
{
  "bits": 392292856,
  "block_height": 510000,
  "merkle_root": "297cfcc6a66e063692b20650d21cc0ac7a2a80f7277ebd7c5d6c7010a070d25c",
  "nonce": 3347656422,
  "prev_block_hash": "0000000000000000002292de0d9f03dfa15a04dbf09102d5d4552117b717fa86
↪",
  "timestamp": 1519083654,
  "version": 536870912
}
```

blockchain.block.get_chunk

Return a concatenated chunk of block headers from the main chain. Typically, a chunk consists of a fixed number of block headers over which difficulty is constant, and at the end of which difficulty is retargeted.

In the case of Bitcoin a chunk is 2,016 headers, each of 80 bytes, so chunk 5 consists of the block headers from height 10,080 to 12,095 inclusive. When encoded as hexadecimal, the result string is twice as long, so for Bitcoin it takes 322,560 bytes, making this a bandwidth-intensive request.

Signature

`blockchain.block.get_chunk`(*index*)

Deprecated since version 1.2: removed in version 1.4

index

The zero-based index of the chunk, an integer.

Result

The binary block headers as hexadecimal strings, in-order and concatenated together. As many as headers as are available at the implied starting height will be returned; this may range from zero to the coin-specific chunk size.

server.version

Identify the client to the server and negotiate the protocol version.

Signature

Changed in version 1.1: *protocol_version* is not ignored.

Changed in version 1.2: Use *server.ping()* rather than sending version requests as a ping mechanism.

Changed in version 1.4: Only the first *server.version()* message is accepted.

- *client_name*

  A string identifying the connecting client software.

- *protocol_version*

  An array `[protocol_min, protocol_max]`, each of which is a string. If `protocol_min` and `protocol_max` are the same, they can be passed as a single string rather than as an array of two strings, as for the default value.

The server should use the highest protocol version both support:

```
version = min(client.protocol_max, server.protocol_max)
```

If this is below the value:

```
max(client.protocol_min, server.protocol_min)
```

then there is no protocol version in common and the server must close the connection. Otherwise it should send a response appropriate for that protocol version.

Result

An array of 2 strings:

```
[server_software_version, protocol_version]
```

identifying the server and the protocol version that will be used for future communication.

Protocol version 1.0: A string identifying the server software.

Examples:

```
server.version("Electrum 3.0.6", ["1.1", "1.2"])
server.version("2.7.1", "1.0")
```

Example Results:

```
["ElectrumX 1.2.1", "1.2"]
"ElectrumX 1.2.1"
```

4.7.5 Protocol Ideas

Note: This is a draft of ideas for a future protocol tentatively called 2.0; they are not implemented and it is likely they will change and that protocol 2.0 will be quite different.

This protocol version makes changes intended to allow clients and servers to more easily scale to support queries about busy addresses. It has changes to reduce the amount of round-trip queries made in common usage, and to make results more compact to reduce bandwidth consumption.

RPC calls with potentially large responses have pagination support, and the return value of *blockchain.scripthash.subscribe()* changes. Script hash *status* had to be recalculated with each new transaction and was undefined if it included more than one mempool transaction. Its calculation is linear in history length resulting in quadratic complexity as history grows. Its calculation for large histories was demanding for both the server to compute and the client to check.

RPC calls and notifications that combined the effects of the mempool and confirmed history are removed.

The changes are beneficial to clients and servers alike, but will require changes to both client-side and server-side logic. In particular, the client should track what block (by hash and height) wallet data is synchronized to, and if that hash is no longer part of the main chain, it will need to remove wallet data for blocks that were reorganized away and get updated information as of the first reorganized block. The effects are limited to script hashes potentially affected by the reorg, and for most clients this will be the empty set.

blockchain.scripthash.subscribe

Subscribe to a script hash.

Signature

`blockchain_.scripthash.`**`subscribe`**`(`*scripthash*`)`

scripthash

The script hash as a hexadecimal string.

Result

Changed in version 2.0.

As of protocol 2.0, the transaction hash of the last confirmed transaction in blockchain order, or `null` if there are none.

For protocol versions 1.4 and below, the *status* of the script hash.

Notifications

Changed in version 2.0.

As this is a subscription, the client receives notifications when the confirmed transaction history and/or associated mempool transactions change.

As of protocol 2.0, the initial mempool and subsequent changes to it are sent with *mempool.changes()* notifications. When confirmed history changes, a notification with signature

blockchain.scripthash.**subscribe**(*scripthash*, *tx_hash*)

is sent, where *tx_hash* is the hash of the last confirmed transaction in blockchain order.

blockchain.scripthash.history

Return part of the confirmed history of a *script hash*.

Signature

blockchain.scripthash.**history**(*scripthash*, *start_height*)

scripthash

The script hash as a hexadecimal string.

start_height

History will be returned starting from this height, a non-negative integer. If there are several matching transactions in a block, the server will return *all* of them – partial results from a block are not permitted. The client can start subsequent requests at one above the greatest returned height and avoid repeats.

Result

A dictionary with the following keys.

- *more*

  `true` indicates that there *may* be more history available. A follow-up request is required to obtain any. `false` means all history to blockchain's tip has been returned.

- *history*

  A list of transactions. Each transaction is itself a list of two elements:

  1. The block height
  2. The transaction hash

Result Examples

```
{
  "more": false,
  "history": [
    [
      200004,
      "acc3758bd2a26f869fcc67d48ff30b96464d476bca82c1cd6656e7d506816412"
    ],
    [
      215008,
      "f3e1bf48975b8d6060a9de8884296abb80be618dc00ae3cb2f6cee3085e09403"
    ]
  ]
}
```

blockchain.scripthash.utxos

Return some confirmed UTXOs sent to a script hash.

Signature

blockchain.scripthash.**utxos**(*scripthash*, *start_height*)

New in version 2.0.

scripthash

The script hash as a hexadecimal string.

start_height

UTXOs will be returned starting from this height, a non-negative integer. If there are several UTXOs in one block, the server will return *all* of them – partial results from a block are not permitted. The client can start subsequent requests at one above the greatest returned height and avoid repeats.

Note: To get the effects of transactions in the mempool adding or removing UTXOs, a client must *blockchain.scripthash.subscribe()* and track mempool transactions sent via *mempool.changes()* notifications.

Result

A dictionary with the following keys.

- *more*

  `true` indicates that there *may* be more UTXOs available. A follow-up request is required to obtain any. `false` means all UTXOs to the blockchain's tip have been returned.

- *utxos*

  A list of UTXOs. Each UTXO is itself a list with the following elements:

  1. The height of the block the transaction is in
  2. The transaction hash as a hexadecimal string
  3. The zero-based index of the output in the transaction's outputs
  4. The output value, an integer in minimum coin units (satoshis)

Result Example

:: TODO

blockchain.transaction.get

Return a raw transaction.

Signature

blockchain_.transaction.**get**(*tx_hash*, *verbose=false*, *merkle=false*)

Changed in version 1.1: ignored argument *height* removed

Changed in version 1.2: *verbose* argument added

Changed in version 2.0: *merkle* argument added

tx_hash

The transaction hash as a hexadecimal string.

verbose

Whether a verbose coin-specific response is required.

merkle

Whether a merkle branch proof should be returned as well.

Result

If *verbose* is `false`:

If *merkle* is `false`, the raw transaction as a hexadecimal string. If `true`, the dictionary returned by *blockchain.transaction.get_merkle()* with an additional key:

hex

The raw transaction as a hexadecimal string.

If *verbose* is `true`:

The result is a coin-specific dictionary – whatever the coin daemon returns when asked for a verbose form of the raw transaction. If *merkle* is `true` it will have an additional key:

merkle

The dictionary returned by *blockchain.transaction.get_merkle()*.

mempool.changes

A notification that indicates changes to unconfirmed transactions of a *subscribed script hash*. As its name suggests the notification is stateful; its contents are a function of what was sent previously.

Signature

`mempool.`**`changes`**(*scripthash*, *new*, *gone*)

New in version 2.0.

The parameters are as follows:

- *scripthash*

  The script hash the notification is for, a hexadecimal string.

- *new*

  A list of transactions in the mempool that have not previously been sent to the client, or whose *confirmed input* status has changed. Each transaction is an ordered list of 3 items:

  1. The raw transaction or its hash as a hexadecimal string. The first time the server sends a transaction it sends it raw. Subsequent references in the same *new* list or in later notifications will send the hash only. Transactions cannot be 32 bytes in size so length can be used to distinguish.
  2. The transaction fee, an integer in minimum coin units (satoshis)
  3. `true` if all inputs are confirmed otherwise `false`

- *gone*

  A list of hashes of transactions that were previously sent to the client as being in the mempool but no longer are. Those transactions presumably were confirmed in a block or were evicted from the mempool.

Notification Example

:: TODO

4.8 Peer Discovery

This was implemented in ElectrumX as of version 0.11.0. Support for IRC peer discovery was removed in ElectrumX version 1.2.1.

The *peer database* is an in-memory store of peers with at least the following information about a peer, required for a response to the `server.peers.subscribe()` RPC call:

- host name
- ip address
- TCP and SSL port numbers
- protocol version
- pruning limit, if any

4.8.1 Hard-coded Peers

A list of hard-coded, well-known peers seeds the peer discovery process. Ideally it should have at least 4 servers that have shown commitment to reliable service.

In ElectrumX this is a per-coin property in lib/coins.py.

4.8.2 server.peers.subscribe

`server.peers.subscribe()` is used by Electrum clients to get a list of peer servers, in preference to a hard-coded list of peer servers in the client, which it will fall back to if necessary.

The server should craft its response in a way that reduces the effectiveness of server sybil attacks and peer spamming.

The response should only include peers it has successfully connected to recently. Only reporting recent good peers ensures that those that have gone offline will be forgotten quickly and not be passed around for long.

In ElectrumX, "recently" is taken to be the last 24 hours. Only one peer from each IPv4/16 netmask is returned, and the number of onion peers is limited.

4.8.3 Maintaining the Peer Database

In order to keep its peer database up-to-date and fresh, after some time has passed since the last successful connection to a peer, an Electrum server should make another attempt to connect, choosing either the TCP or SSL port.

On connecting it should issue `server.peers.subscribe()`, `blockchain.headers.subscribe()`, and `server.features()` RPC calls to collect information about the server and its peers. If the peer seems to not know of you, you can issue a `server.add_peer()` call to advertise yourself. Once this is done and replies received, terminate the connection.

The peer database should view information obtained from an outgoing connection as authoritative, and prefer it to information obtained from any other source.

On connecting, a server should confirm the peer is serving the same network, ideally via the genesis block hash of the `server.features()` RPC call below. Also the height reported by the peer should be within a small number of the expected value. If a peer is on the wrong network it should never be advertised to clients or other peers. Such invalid peers should perhaps be remembered for a short time to prevent redundant revalidation if other peers persist in advertising them, and later forgotten.

If a connection attempt fails, subsequent reconnection attempts should follow some kind of exponential backoff.

If a long period of time has elapsed since the last successful connection attempt, the peer entry should be removed from the database. This ensures that all peers that have gone offline will eventually be forgotten by the network entirely.

ElectrumX will connect to the SSL port if both ports are available. If that fails it will fall back to the TCP port. It tries to reconnect to a good peer at least once every 24 hours, and a failing after 5 minutes but with exponential backoff. It forgets a peer entirely if a few days have passed since a successful connection. ElectrumX attempts to connect to onion peers through a Tor proxy that can be configured or that it will try to autodetect.

4.8.4 server.features

`server.features()` is a fairly new RPC call that a server can use to advertise what services and features it offers. It is intended for use by Electrum clients as well as other peers. Peers will use it to gather peer information from the peer itself.

The call takes no arguments and returns a dictionary keyed by feature name whose value gives details about the feature where appropriate. If a key is missing the feature is presumed not to be offered.

4.8.5 server.add_peer

`server.add_peer()` is intended for a new server to get itself in the connected set.

A server receiving a `server.add_peer()` call should not replace existing information about the host(s) given, but instead schedule a separate connection to verify the information for itself.

To prevent abuse a server may do nothing with second and subsequent calls to this method from a single connection.

The result should be True if accepted and False otherwise.

4.8.6 Notes for Implementors

- it is very important to only accept peers that appear to be on the same network. At a minimum the genesis hash should be compared (if the peer supports `server.features()`), and also that the peer's reported height is within a few blocks of your own server's height.
- care should be taken with the `server.add_peer()` call. Consider only accepting it once per connection. Clearnet peer requests should check the peer resolves to the requesting IP address, to prevent attackers from being able to trigger arbitrary outgoing connections from your server. This doesn't work for onion peers so they should be rate-limited.
- it should be possible for a peer to change their port assignments - presumably connecting to the old ports to perform checks will not work.
- peer host names should be checked for validity before accepting them; and *localhost* should probably be rejected. If it is an IP address it should be a normal public one (not private, multicast or unspecified).
- you should limit the number of new peers accepted from any single source to at most a handful, to limit the effectiveness of malicious peers wanting to trigger arbitrary outgoing connections or fill your peer tables with junk data.
- in the response to `server.peers.subscribe()` calls, consider limiting the number of peers on similar IP subnets to protect against sybil attacks, and in the case of onion servers the total returned.
- you should not advertise a peer's IP address if it also advertises a hostname (avoiding duplicates).

4.9 RPC Interface

You can query the status of a running server, and affect its behaviour by sending **JSON RPC** commands to the LocalRPC port it is listening on. This is best done using the `electrumx_rpc` script provided.

The general form of invocation is:

```
electrumx_rpc [-p PORT] <command> [arg1 [arg2...]]
```

The port to send the commands to can be specified on the command line, otherwise the environment variable `RPC_PORT` is used, and if that is not set then **8000** is assumed.

The following commands are available:

4.9.1 add_peer

Add a peer to the peers list. ElectrumX will schedule an immediate connection attempt. This command takes a single argument: the peer's "real name" as it used to advertise itself on IRC:

```
$ electrumx_rpc add_peer "ecdsa.net v1.0 s110 t"
"peer 'ecdsa.net v1.0 s110 t' added"
```

4.9.2 daemon_url

This command takes an optional argument that is interpreted identically to the *DAEMON_URL* environment variable. If omitted, the default argument value is the process's existing *DAEMON_URL* environment variable.

This command replaces the daemon's URL at run-time, and also forcefully rotates to the first URL in the list.

For example, in case ElectrumX has previously failed over to a secondary daemon and you want to revert to the primary having resolved the connectivity issue, invoking this command without an argument will have that effect.

4.9.3 disconnect

Disconnect the given session IDs or group names.

Session IDs can be obtained in the logs or with the *sessions* RPC command. Group names can be obtained with the *groups* RPC command.

The special string `all` disconnects all sessions.

Example:

```
$ electrumx_rpc disconnect 209.59.102 34 2
[
    "disconnecting session 34",
    "disconnecting group 209.59.102"
    "unknown: 2",
]
```

4.9.4 getinfo

Return a summary of server state. This command takes no arguments. A typical result is as follows (with annotated comments):

```
$ electrumx_rpc getinfo
{
    "coin": "BitcoinSegwit",
    "daemon": "127.0.0.1:9334/",
    "daemon height": 572154,         # The daemon's height when last queried
    "db height": 572154,             # The height to which the DB is flushed
    "groups": 586,                   # The number of session groups
    "history cache": "185,014 lookups 9,756 hits 1,000 entries",
    "merkle cache": "280 lookups 54 hits 213 entries",
    "peers": {                       # Peer information
        "bad": 1,
        "good": 51,
        "never": 2,
        "stale": 0,
        "total": 54
    },
    "pid": 11804,                    # Process ID
    "request counts": {              # Count of RPC requests by method name
        "blockchain.block.header": 245,
        "blockchain.block.headers": 70,
        "blockchain.headers.subscribe": 2825,
        "blockchain.scripthash.get_history": 196,
        "blockchain.scripthash.subscribe": 184626,
        "blockchain.transaction.broadcast": 19,
        "blockchain.transaction.get": 213,
        "blockchain.transaction.get_merkle": 289,
        "getinfo": 3,
        "groups": 1,
        "server.add_peer": 9,
        "server.banner": 740,
        "server.donation_address": 754,
        "server.features": 50,
        "server.peers.subscribe": 792,
        "server.ping": 6412,
        "server.version": 2866
    },
    "request total": 216820,         # Total requests served
    "sessions": {                    # Live session stats
        "count": 670,
        "count with subs": 45,
        "errors": 0,
        "logged": 0,
        "paused": 0,
        "pending requests": 79,      # Number of requests currently being processed
        "subs": 36292                # Total subscriptions
    },
    "tx hashes cache": "289 lookups 38 hits 213 entries",
    "txs sent": 19,                  # Transactions broadcast
    "uptime": "01h 39m 04s",
    "version": "ElectrumX 1.10.1"
}
```

Each ill-formed request, or one that does not follow the Electrum protocol, increments the error count of the session that sent it.

logging of sessions can be enabled by RPC.

For more information on peers see *here*.

Clients that are slow to consume data sent to them are *paused* until their socket buffer drains sufficiently, at which point processing of requests resumes.

Apart from very short intervals, typically after a new block or when a client has just connected, the number of unprocessed requests should be low, say 250 or fewer. If it is over 1,000 the server is overloaded.

Sessions are put into groups, primarily as an anti-DoS measure. Currently each session goes into two groups: one for an IP subnet, and one based on the timeslice it connected in. Each member of a group incurs a fraction of the costs of the other group members. This appears in the *sessions_* list under the column XCost.

4.9.5 groups

Return a list of all current session groups. Takes no arguments.

The output is quite similar to the *sessions* command.

4.9.6 log

Toggle logging of the given session IDs or group names. All incoming requests for a logged session are written to the server log. The arguments are case-insensitive.

When a group is specified, logging is toggled for its current members only; there is no effect on future group members.

Session IDs can be obtained in the logs or with the *sessions* RPC command. Group names can be obtained with the *groups* RPC command.

The special string `all` turns on logging of all current and future sessions, `none` turns off logging of all current and future sessions, and `new` toggles logging of future sessions.

Example:

```
$ electrumx_rpc log new 6 t0 z
[
  "logging new sessions",
  "logging session 6",
  "logging session 3",
  "logging session 57",
  "logging session 12"
  "unknown: z",
]
```

In the above command sessions 3, 12 and 57 were in group *t0* (in fact, session 6 was too).

4.9.7 peers

Return a list of peer Electrum servers serving the same coin network. This command takes no arguments.

Peer data is obtained via a peer discovery protocol documented *here*:

```
$ electrumx_rpc peers
Host                          Status   TCP   SSL Server             Min  Max ␣
↪Pruning   Last Good    Last Try Tries              Source IP Address
bch.tedy.pw                   good   50001 50002 ElectrumX 1.2.1    0.9  1.2       ␣
↪ 07h 29m 23s 07h 30m 40s     0                 peer 185.215.224.26
shsmithgoggryfbx.onion        good   60001 60002 ElectrumX 1.2.1    0.9  1.2       ␣
↪ 07h 30m 34s 07h 30m 38s     0                 peer
```

(continues on next page)

(continued from previous page)

```
bccarihace4jdcnt.onion          good    52001 52002 ElectrumX 1.2.1     0.9  1.2
↪ 07h 30m 34s 07h 30m 39s       0                   peer
[...]
electroncash.checksum0.com      good    50001 50002 ElectrumX 1.2.1     0.9  1.1
↪ 07h 30m 40s 07h 30m 41s       0                   peer 149.56.198.233
```

4.9.8 query

Run a query of the UTXO and history databases against one or more addresses, hex scripts or ASCII names (for coins that have an index on names like Namecoin). *–limit <N>* or *-l <N>* limits the output for each kind to that many entries. History is printed in blockchain order; UTXOs in an arbitrary order.

For example:

```
$ electrumx_rpc query --limit 5 76a91462e907b15cbf27d5425399ebf6f0fb50ebb88f1888ac
Script: 76a91462e907b15cbf27d5425399ebf6f0fb50ebb88f1888ac
History #1: height 123,723 tx_hash␣
↪3387418aaddb4927209c5032f515aa442a6587d6e54677f08a03b8fa7789e688
History #2: height 127,280 tx_hash␣
↪4574958d135e66a53abf9c61950aba340e9e140be50efeea9456aa9f92bf40b5
History #3: height 127,909 tx_hash␣
↪8b960c87f9f1a6e6910e214fcf5f9c69b60319ba58a39c61f299548412f5a1c6
History #4: height 127,943 tx_hash␣
↪8f6b63012753005236b1b76e4884e4dee7415e05ab96604d353001662cde6b53
History #5: height 127,943 tx_hash␣
↪60ff2dfdf67917040139903a0141f7525a7d152365b371b35fd1cf83f1d7f704
UTXO #1: tx_hash 9aa497bf000b20f5ec5dc512bb6c1b60b68fc584d38b292b434e839ea8807bf0 tx_
↪pos 0 height 254,148 value 5,500
UTXO #2: tx_hash 1c998142a5a5aae6f8c1eab245351413fe8d4032a3f14345f9943a0d0bc90ec0 tx_
↪pos 0 height 254,161 value 5,500
UTXO #3: tx_hash 53345491b4829140be53f30079c6e4556a18545343b122900ebbfa158f9ca97a tx_
↪pos 0 height 254,163 value 5,500
UTXO #4: tx_hash c71ad947ac46af217da3cd5521113cbd03e36ddada2b4452afe6c15f944d2529 tx_
↪pos 0 height 372,916 value 1,000
UTXO #5: tx_hash c944a6acac054275a5e294e746d9ce79f6dcae91f3b4f5a84561aee6404a55b3 tx_
↪pos 0 height 254,148 value 5,500
Balance: 17.8983303 BCH
```

4.9.9 reorg

Force a block chain reorganisation, primarily for debugging purposes. This command takes an optional argument - the number of blocks to reorg - which defaults to 3.

That number of blocks will be backed up - using undo information stored in ElectrumX's database - and then ElectrumX will move forwards on the daemon's main chain to its current height.

4.9.10 sessions

Return a list of all current sessions. Takes no arguments:

```
ID      Flags            Client Proto    Cost   XCost  Reqs   Txs    Subs    Recv Recv␣
↪KB     Sent Sent KB      Time                   Peer
1       S6                1.1.1   1.4       0      16     0      0       0       3
↪ 0        3        0    05m42s 165.255.191.213:22349
2       S6       all_seeing_eye   1.4       0      16     0      0       0       2
↪ 0        2        0    05m40s   67.170.52.226:24995
4       S6                3.3.2   1.4       0      16     0      0      34      45
↪ 5       45        3    05m40s 185.220.100.252:40463
3       S6                1.1.2   1.4       0      16     0      0       0       3
↪ 0        3        0    05m40s    89.17.142.28:59241
```

The columns show information by session: the session ID, flags (see below), how the client identifies itself - typically the Electrum client version, the protocol version negotiated, the session cost, the additional session cost accrued from its groups, the number of unprocessed requests, the number of transactions sent, the number of address subscriptions, the number of requests received and their total size, the number of messages sent and their size, how long the client has been connected, and the client's IP address (if anonymous logging is disabled).

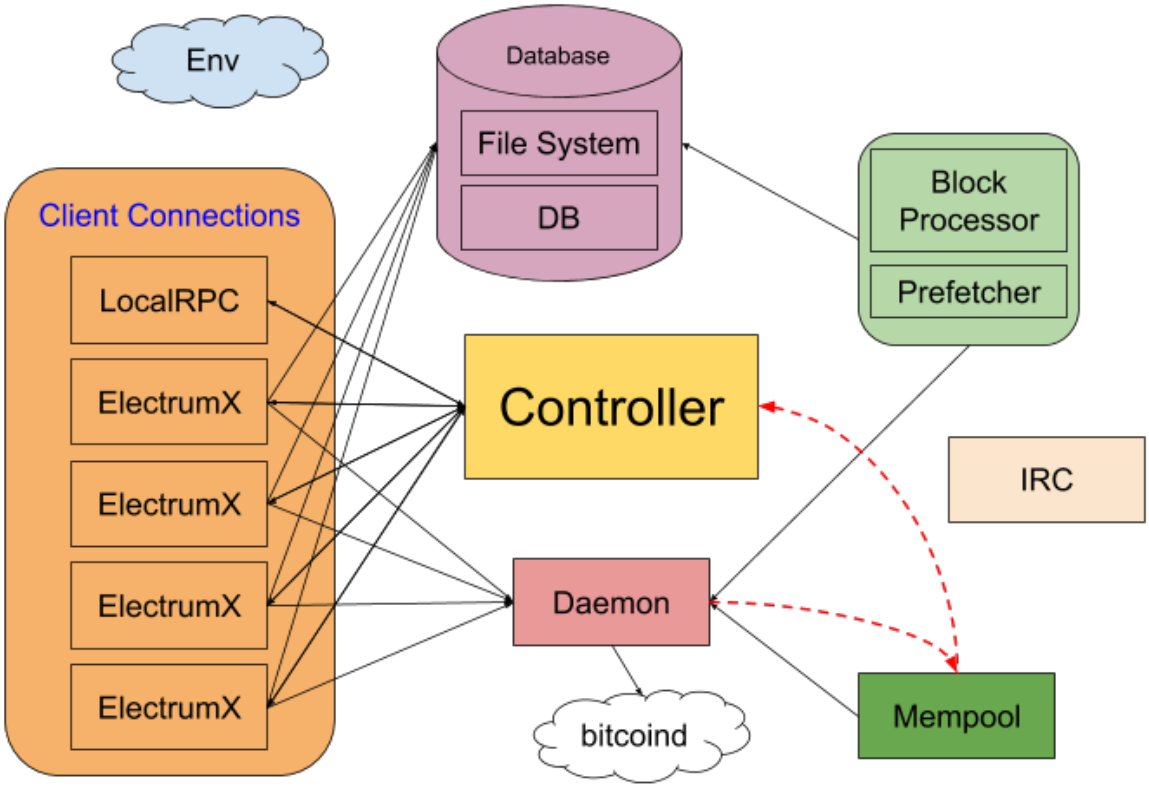
The flags are:

- `S` an SSL connection
- `T` a TCP connection
- `R` a local RPC connection
- `L` a logged session
- `C` a connection that is being closed
- the non-negative number is the connection "cost", with lower numbers having higher priority. RPC connections have cost `0`, normal connections have cost at least `1`.

4.9.11 stop

Flush all cached data to disk and shut down the server cleanly, as if sending the *KILL* signal. Be patient - during initial sync flushing all cached data to disk can take several minutes. This command takes no arguments.

4.10 Architecture



4.10.1 Env

Holds configuration taken from the environment, with apprioriate defaulting appropriately. Generally passed to the constructor of other components which take their settings from it.

4.10.2 Controller

The central part of the server process initialising and coordinating all the others. Manages resource usage.

4.10.3 LocalRPC

Handles local JSON RPC connections querying ElectrumX server state. Started when the ElectrumX process starts.

4.10.4 ElectrumX

Handles JSON Electrum client connections over TCP or SSL. One instance per client session. Should be the only component concerned with the details of the Electrum wire protocol.

Not started until the Block Processor has caught up with bitcoind.

4.10.5 Daemon

Encapsulates the RPC wire protocol with bitcoind for the whole server. Transparently handles temporary bitcoind connection errors, and fails over if necessary.

Notifies the Mempool when the list of mempool transaction hashes is updated.

4.10.6 Block Processor

Responsible for managing block chain state (UTXO set, history, transaction and undo information) and for handling block chain reorganisations.

When caught up, processes new blocks as they are found, and flushes the updates to the Database immediately.

When syncing uses caches for in-memory state updates since the prior flush. Occasionally flushes state to the storage layer when caches get large.

4.10.7 Prefetcher

Cooperates with the Block Processor to asynchronously prefetch blocks from bitcoind. Once it has caught up it additionally asks the Daemon to refresh its view of bitcoind's mempool transaction hashes. Serves blocks to the Block Processor via a queue.

4.10.8 Mempool

Handles all the details of maintaining a representation of bitcoind's mempool state. Obtains the list of current mempool transaction hashes from the Daemon when notified by the Prefetcher.

Notifies the Controller that addresses have been touched when the mempool refreshes (or implicitly when a new block is found).

4.10.9 Database

The underlying data store, made up of the DB backend (such as *leveldb*) and the host filesystem.

4.11 Authors

- Neil Booth

  Creator and maintainer.

- Johann Bauer

  Backend DB abstraction.

- John Jegutanis

  Alt-chain integrations.

CHAPTER 5

Indices and tables

- genindex
- search

Index

I

L

M

N

P

R

S