Fulcrum Documentation

Release Fulcrum Protocol Reference 1.4.5

Neil Booth, Calin Culianu

Mar 02, 2021

Contents

The Fulcrum protocol is basically the same as ElectrumX and other Electrum-based protocol servers such ElectrsCash. This protocol was initially designed for Bitcoin BTC but has since been adopted by Bitcoin Cash for use with Electron Cash, and continues to evolve independently of BTC. The Bitcoin Cash variant of the protocol is known as the Electrum Cash Protocol.

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

CHAPTER 1

Protocol Basics

1.1 Message Stream

Clients and servers communicate using **JSON RPC** over an unspecified underlying stream transport. Examples include TCP and SSL (with WS and WSS support planned for the future).

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 *not implemented* in Fulcrum.

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. Fulcrum does and will expect clients advertizing so to function correctly. Those that do not will be disconnected and possibly blacklisted.

For the TCP and SSL transports: Each RPC call MUST be delimited by a single newline. 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.

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.

1.2 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.

1.3 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.

1.4 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.

1.5 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.

1.6 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 BTC and BCH chains 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. Fulcrum implements an optimization such that it hashes only approximately 180KB of data per proof.

1.7 Double Spend Proofs (dsproofs)

A double spend proof is information collected by the Bitcoin Cash peer-to-peer network on transaction inputs for transactions in the mempool that are seen to have been attempted at being double-spent. Double-spend proofs only apply to mempool transactions. Once a transaction is confirmed, the double-spend attempt is no longer relevant (since the transaction cannot be double-spent anymore unless there is a reorg). Double-spend proofs indicate that a transaction may not confirm as expected, and that instead there is a risk that its conflicting transaction will confirm instead.

The specification for dsproofs can be found here.

In Fulcrum, the dsproofs are returned as JSON objects with the following keys:

- **dspid**

  This is the hexadecimal hash of the `dsproof` as would be returned by querying the BCHN dsproof RPC `getdsproof`.

- **hex**

  The raw serialized double-spend proof itself.

- **outpoint**

  A JSON object containing the following keys:

  - **txid**

    The transaction hash of the transaction that generated this outpoint.

  - **vout**

    The integer output number for this outpoint.

- **txid**

  The primary transaction that is associated with this `dsproof`.

- **descendants**

  A JSON array of *txid*'s of all the transactions that are potentially affected by this double-spend attempt. This list will include *txid* above plus all of its descendant transactions.

An example *dsproof* object as might be returned by Fulcrum:

```
{
  "dspid": "587d18bf8a64ede9c7450fdaeab27b9b3c46cfa8948f4c145f889601153c56b0",
  "txid": "5b59ce35093fbd13549cd6f203d4b5b01762d70e75b8e9733dfc463e0ff8cc13",
  "hex":
↪"410c56078977120e828e4aacdd813a818d17c47d94183aa176d62c805d47697ddddf46c2ab68ee1e46a3e17aa7da548c
↪",
  "outpoint": {
    "txid": "f6e2a16ba665d5402dad147fe35872961bc6961da62345a2171ee001cfcf7600",
    "vout": 0
  },
  "descendants": [
    "36fbb099e6de59d23477727e3199c65caae35ded957660f56fc681a6d81d5570",
```

(continues on next page)

(continued from previous page)

```
    "5b59ce35093fbd13549cd6f203d4b5b01762d70e75b8e9733dfc463e0ff8cc13"
  ]
}
```

Note that as of March 2021, only servers running Bitcoin Cash Node v22.3.0 or later are capable of reporting double-spend proofs via RPC, and thus only such servers will provide double-spend proofs to clients via the Electrum Cash protocol. Servers that support *dsproof* will have the key `"dsproof"` set to `true` in their *`server.features()`* map.

CHAPTER 2

Protocol Methods

2.1 blockchain.address.get_balance

Return the confirmed and unconfirmed balances of a Bitcoin Cash address.

Signature

`blockchain.address.get_balance(address)`

New in version 1.4.3.

- *address*

  The address as a Cash Address string (with or without prefix, case insensitive). Some server implementations do not support Legacy (base58) addresses and are not required to do so by this specification. However, Fulcrum does support both Legacy and Cash Address encodings.

Result

See `blockchain.scripthash.get_balance()`.

2.2 blockchain.address.get_history

Return the confirmed and unconfirmed history of a Bitcoin Cash address.

Signature

`blockchain.address.get_history(address)`

New in version 1.4.3.

- *address*

  The address as a Cash Address string (with or without prefix, case insensitive). Some server implementations do not support Legacy (base58) addresses and are not required to do so by this specification. However, Fulcrum does support both Legacy and Cash Address encodings.

Result

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

2.3 blockchain.address.get_mempool

Return the unconfirmed transactions of a Bitcoin Cash address.

Signature

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

New in version 1.4.3.

- *address*

  The address as a Cash Address string (with or without prefix, case insensitive). Some server implementations do not support Legacy (base58) addresses and are not required to do so by this specification. However, Fulcrum does support both Legacy and Cash Address encodings.

Result

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

2.4 blockchain.address.get_scripthash

Translate a Bitcoin Cash address to a *script hash*. This method is potentially useful for clients preferring to work with *script hashes* but lacking the local libraries necessary to generate them.

Signature

`blockchain.address.`**`get_scripthash`**`(`*address*`)`

New in version 1.4.3.

- *address*

  The address as a Cash Address string (with or without prefix, case insensitive). Some server implementations do not support Legacy (base58) addresses and are not required to do so by this specification. However, Fulcrum does support both Legacy and Cash Address encodings.

Result

The unique 32-byte hex-encoded *script hash* that corresponds to the decoded address.

2.5 blockchain.address.listunspent

Return an ordered list of UTXOs sent to a Bitcoin Cash address.

Signature

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

New in version 1.4.3.

- *address*

  The address as a Cash Address string (with or without prefix, case insensitive). Some server implementations do not support Legacy (base58) addresses and are not required to do so by this specification. However, Fulcrum does support both Legacy and Cash Address encodings.

Result

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

2.6 blockchain.address.subscribe

Subscribe to a Bitcoin Cash address.

Signature

`blockchain.address.subscribe(address)`

New in version 1.4.3.

address

The address as a Cash Address string (with or without prefix, case insensitive). Some server implementations do not support Legacy (base58) addresses and are not required to do so by this specification. However, Fulcrum does support both Legacy and Cash Address encodings.

Result

The *status* of the address.

Notifications

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

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

Note: The address returned back to the client when notifying of status changes will be in the same encoding and syle as was provided when subscribing. In effect, a whitespace-stripped version of the address string that the client provided will be sent back to the client when notifying, in order to make it easier for clients to track the notification.

It is unspecified what happens if a client subscribes to the same address using multiple encodings or styles, but it is RECOMMENDED that servers simply update their internal subscription tables on subsequent subscriptions to the same destination such that they honor the latest subscription only, and not subscribe clients multiple times to the same logical destination. For example, Fulcrum server will simply update its table for how to refer to the subscription and send clients subsequent notifications using the latest encoding style of that particular address that the client last provided.

Similarly, if a client mixes *blockchain.address.** and *blockchain.scripthash.** calls to the server, it is RECOMMENDED that the server treat all addresses as equivalent to their scripthashes internally such that it is possible to subscribe by address and later unsubscribe by scripthash, for example.

2.7 blockchain.address.unsubscribe

Unsubscribe from a Bitcoin Cash address, preventing future notifications if its *status* changes.

Signature

`blockchain.address.`**`unsubscribe`**(*address*)

New in version 1.4.3.

address

The address as a Cash Address string (with or without prefix, case insensitive). Some server implementations do not support Legacy (base58) addresses and are not required to do so by this specification. However, Fulcrum does support both Legacy and Cash Address encodings.

Result

Returns `true` if the address was previously 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.

2.8 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.

- *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"
}
```

2.9 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":
↪"0100000000000000000000000000000000000000000000000000000000000000000000003ba3edfd7a7b12b27ac72c3e6
↪"
  "max": 2016
}
```

2.10 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*`)`

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

```
0.00101079
```

2.11 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":
↪"00000020890208a0ae3a3892aa047c5468725846577cfcd9b512b5000000000000000000005dc2b02f2d297a9064ee10303
↪"
}
```

Notifications

As this is a subcription, 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.

2.12 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`**`()`

Result

The fee in whole coin units (BTC, not satoshis for Bitcoin) as a floating point number.

Example Results

```
1e-05
```

```
0.0
```

2.13 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 satoshis.

Result Example

```
{
  "confirmed": 103873966,
  "unconfirmed": 236844
}
```

2.14 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"
```

(continues on next page)

(continued from previous page)

```
  }
]
```

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

2.15 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
  }
]
```

2.16 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
  }
]
```

2.17 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*)

2.18 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 previously 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.

2.19 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"
```

2.20 blockchain.transaction.dsproof.get

Returns information on a *double-spend proof*. The query can be by either *tx_hash* or *dspid*.

Signature

`blockchain.transaction.dsproof.get`(*hash*)

New in version 1.4.5.

hash

The transaction hash (or *dspid*) as a hexadecimal string.

Result

If the transaction in question has an associated *dsproof*, then a JSON object. Otherwise `null`.

Example Results:

```
{
  "dspid": "587d18bf8a64ede9c7450fdaeab27b9b3c46cfa8948f4c145f889601153c56b0",
  "txid": "5b59ce35093fbd13549cd6f203d4b5b01762d70e75b8e9733dfc463e0ff8cc13",
  "hex":
↪"410c56078977120e828e4aacdd813a818d17c47d94183aa176d62c805d47697ddddf46c2ab68ee1e46a3e17aa7da548c3
↪",
  "outpoint": {
    "txid": "f6e2a16ba665d5402dad147fe35872961bc6961da62345a2171ee001cfcf7600",
    "vout": 0
  },
  "descendants": [
    "36fbb099e6de59d23477727e3199c65caae35ded957660f56fc681a6d81d5570",
    "5b59ce35093fbd13549cd6f203d4b5b01762d70e75b8e9733dfc463e0ff8cc13"
  ]
}
```

2.21 blockchain.transaction.dsproof.list

List all of the transactions that currently have double-spend proofs associated with them.

Signature

`blockchain.transaction.dsproof.list`()

New in version 1.4.5.

Result

A JSON array of hexadecimal strings. May be empty. Each string is a transaction hash of an in-mempool transaction that has a double-spend proof associated with it. Each of the hashes appearing in the list may be given as an argument to *`blockchain.transaction.dsproof.get()`* in order to obtain the associated double-spend proof for that transaction.

Example Results:

```
[
  "e67cc122f3c28a4243c3a1b14b38a9474c22ba928af9a194ca2b85426f0fd1bb",
  "077f0cc2439f2e48567c72eeeba5a447f8649c00c3d18ab6516eccfd4119726f",
  "ccc2f0d90b7067a83566024d4df842f0b6cb8180e18d642fcc85cae8acadbd58"
]
```

2.22 blockchain.transaction.dsproof.subscribe

Subscribe for *dsproof* notifications for a transaction.

Signature

`blockchain.transaction.dsproof.`**`subscribe`**`(`*tx_hash*`)`

New in version 1.4.5.

tx_hash

The transaction hash as a hexadecimal string.

Result

A result identical to what one would get from `blockchain.transaction.dsproof.get()`.

Notifications

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

`blockchain.transction.dsproof.`**`subscribe`**`(`*tx_hash*, *dsproof*`)`
With *dsproof* being identical to what one would get from invoking `blockchain.transaction.dsproof.get()` for that particular *tx_hash*.

2.23 blockchain.transaction.dsproof.unsubscribe

Unsubscribe from receiving any further *dsproof* notifications for a transaction.

Signature

`blockchain.transaction.dsproof.`**`unsubscribe`**`(`*tx_hash*`)`

New in version 1.4.5.

tx_hash

The transaction hash as a hexadecimal string.

Result

Returns `true` if the transaction was previously subscribed-to for dsproof notifications, otherwise `false`. Note that `false` might be returned even for something subscribed-to earlier, because the server can drop subscriptions in rare circumstances.

2.24 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": "0000000000000000015a4f37ece911e5e3549f988e855548ce7494a0a08b2ad6",
  "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,
```

(continues on next page)

(continued from previous page)

```
                                   "type": "pubkeyhash"},
                 "value": 0.0856647},
               { "n": 1,
                 "scriptPubKey": { "addresses": [ "17NMgYPrguizvpJmB1Sz62ZHeeFydBYbZJ"],
                                   "asm": "OP_DUP OP_HASH160␣
↪45dac110239a7a3814535c15858b939211f85298 OP_EQUALVERIFY OP_CHECKSIG",
                                   "hex":
↪"76a91445dac110239a7a3814535c15858b939211f8529888ac",
                                   "reqSigs": 1,
                                   "type": "pubkeyhash"},
                 "value": 0.1360904}]}
```

2.25 blockchain.transaction.get_height

Returns the block height for a confirmed transaction, or 0 for a mempool transaction, given its hash.

Signature

`blockchain.transaction.`**`get_height`**`(`*tx_hash*`)`

New in version 1.4.5.

tx_hash

The transaction hash as a hexadecimal string.

Result

Either a numeric value or `null`.

- Numeric values `> 0`

  The transaction is confirmed at this block height.

- Numeric values `== 0`

  The transaction is not confirmed but is in the mempool.

- `null`

  The transaction is unknown.

2.26 blockchain.transaction.get_merkle

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

Signature

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

Changed in version 1.4.5: *height* is no longer required and is now optional

tx_hash

The transaction hash as a hexadecimal string.

height (optional in 1.4.5 or above)

The height at which it was confirmed, an integer. As of version 1.4.5 of the protocol, this second argument may be omitted, however if it is included the lookup may return the results slightly faster since an extra database lookup is avoided on the server-side.

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
}
```

2.27 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"
  ]
}
```

2.28 blockchain.transaction.subscribe

Subscribe to a transaction in order to receive future notifications if its confirmation status changes.

Signature

`blockchain.transaction.`**`subscribe`**`(`*tx_hash*`)`

New in version 1.4.5.

tx_hash

The transaction hash as a hexadecimal string.

Result

A result identical to what one would get from *`blockchain.transaction.get_height()`*.

Notifications

The client will receive a notification when the confirmation status of the transaction changes. Its signature is

`blockchain.transction.`**`subscribe`**`(`*tx_hash*, *height*`)`

With *height* being identical to what one would get from invoking *`blockchain.transaction.get_height()`*.

2.29 blockchain.transaction.unsubscribe

Unsubscribe from a transaction, preventing future notifications if its confirmation status changes.

Signature

`blockchain.transaction.`**`unsubscribe`**`(`*tx_hash*`)`

New in version 1.4.5.

tx_hash

The transaction hash as a hexadecimal string.

Result

Returns `true` if the transaction was previously 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.

2.30 blockchain.utxo.get_info

Return information for an unspent transaction output.

Signature

`blockchain.utxo.`**`get_info`**`(`*tx_hash*, *out_n*`)`

New in version 1.4.4.

tx_hash

The UTXO's transaction hash as a hexadecimal string.

out_n

The UTXO's transaction output number. This should be a number in the range 0 <= out_n <= 65535

Result

If the UTXO in question does not exist or is spent, `null` is returned. Otherwise, a dictionary will be returned containing the following keys:

- *confirmed_height*

  (Optional) The integer height of the block the UTXO's transaction was confirmed in. This key will be missing from the dictionary if the transaction is in the mempool and is not yet confirmed.

- *scripthash*

  The output's destination *script hash* as a hexadecimal string.

- *value*

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

Result Example

```
{
  "confirmed_height": 602123,
  "scripthash": "1c1e184c97abc87626c497b95f755df1025f48b9c27d037ea335677c57f38e5c",
  "value": 45318048
}
```

2.31 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.

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 $\text{fee}_{n-1} > \text{fee}_n$.

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]]
```

2.32 server.add_peer

A newly-started server uses this call to get itself into other servers' peers lists. It sould 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.

2.33 server.banner

Return a banner to be shown in the Electron Cash console.

Signature

`server.banner()`

Result

A string.

Example Result

```
"Welcome to Fulcrum!"
```

2.34 server.donation_address

Return a server donation address.

Signature

`server.donation_address()`

Result

A string.

Example Result

```
"1BWwXJH3q6PRsizBkSGm2Uw4Sz1urZ5sCj"
```

2.35 server.features

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

Signature

`server.features()`

Changed in version 1.4.2: *hosts* key is no longer required, but recommended.

Changed in version 1.4.5: *dsproof* key added (optional).

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.

- *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 result to the `server.version()` RPC call.

- *protocol_max*
- *protocol_min*

  Strings that are the minimum and maximum Electrum Cash 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.

The following features are RECOMMENDED that be reported by the servers.

- *hosts*

  A dictionary, keyed by host name, that this server can be reached at. If this dictionary is missing, then this is a way to signal to other servers that while this host is reachable, it does not wish to peer with other servers. A server SHOULD stop peering with a peer if it sees the *hosts* dictionary for its peer is empty and/or no longer contains the expected route (e.g. hostname). Normally this dictionary will only contain 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.

  – *ws_port*

    An integer. Omit or set to `null` if Web Socket (ws://) connectivity is not provided.

  – *wss_port*

    An integer. Omit or set to `null` if Web Socket Secure (wss://) connectivity is not provided.

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

- *dsproof*

  A boolean value. If present and set to `true`, then the server has *double-spend proof* support, and it supports the `blockchain.transaction.dsproof.*` set of RPC methods. If this key is missing or `false`, then the server does not support *dsproofs*.

Example Result

```
{
    "genesis_hash": "000000000933ea01ad0ee984209779baaec3ced90fa3f408719526f8d77f4943
↪",
    "hosts": {"14.3.140.101": {"tcp_port": 51001, "ssl_port": 51002}},
    "protocol_max": "1.4",
    "protocol_min": "1.4.3",
    "pruning": null,
    "server_version": "Fulcrum 1.0.5",
    "hash_function": "sha256",
    "dsproof": true
}
```

2.36 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.

2.37 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`.

2.38 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("Electron Cash 3.3.6", ["1.2", "1.4"])
```

Example Result:

```
["Fulcrum 1.0.5", "1.4"]
```

CHAPTER 3

Protocol Changes

This documents lists changes made by protocol version.

3.1 Version 1.0

3.1.1 Deprecated methods

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

3.2 Version 1.1

3.2.1 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.

3.2.2 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()`

3.2.3 Removed methods

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

3.3 Version 1.2

3.3.1 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.

3.3.2 New methods

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

3.3.3 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`.

3.4 Version 1.3

3.4.1 Changes

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

3.4.2 New methods

- *`blockchain.block.header()`*

3.4.3 Removed methods

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

3.4.4 Deprecated methods

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

3.5 Version 1.4

This version removes all support for *deserialized headers*.

3.5.1 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.

3.5.2 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.

3.5.3 Removed methods

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

3.6 Version 1.4.1

3.6.1 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.

3.7 Version 1.4.1

3.7.1 New methods

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

3.8 Version 1.4.2

- *server.features()* changed the requirement of key *hosts* from being MUST be present to RECOMMENDED. Note that Fulcrum and ElectrumX will not peer with your server without this key.

3.9 Version 1.4.3

3.9.1 New methods

- *blockchain.address.get_balance()* was brought back after having been removed in 1.3.
- *blockchain.address.get_history()* was brought back after having been removed in 1.3.
- *blockchain.address.get_mempool()* was brought back after having been removed in 1.3.
- *blockchain.address.get_scripthash()* to translate an address into a script hash.
- *blockchain.address.listunspent()* was brought back after having been removed in 1.3.
- *blockchain.address.subscribe()* was brought back after having been removed in 1.3.
- *blockchain.address.unsubscribe()* to unsubscribe from an address.

3.10 Version 1.4.4

3.10.1 New methods

- *blockchain.utxo.get_info()* gets information for an unspent transaction output.

3.11 Version 1.4.5

3.11.1 Changes

- `blockchain.transaction.get_merkle()` now no longer requires the second argument *height*. This argument is now optional but still recommended, in order to save the server from having to look it up.
- `server.features()` added a new optional key, `"dsproof"`.

3.11.2 New methods

- `blockchain.transaction.get_height()` to retrieve the height of a transaction.
- `blockchain.transaction.subscribe()` to be notified of transaction confirmation status changes.
- `blockchain.transaction.unsubscribe()` to unsubscribe from a previously subscribed-to transaction.
- `blockchain.transaction.dsproof.get()` to retrieve information about a double-spend proof associated with a *tx_hash*.
- `blockchain.transaction.dsproof.list()` to list the double-spend proofs currently being tracked in the bitcoin daemon's mempool.
- `blockchain.transaction.dsproof.subscribe()` to receive notifications on whether a particular transaction is known by the network to have been double-spent.
- `blockchain.transaction.dsproof.unsubscribe()` to unsubscribe from receiving dsproof notifications for a particular transaction.

CHAPTER 4

Removed Protocol Methods

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

4.1 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.

4.1.1 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, *re-added in version 1.4.3*

- *address*

  The address as a Base58 string.

Result

See *blockchain.scripthash.get_balance()*.

4.1.2 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, *re-added in version 1.4.3*

- *address*

  The address as a Base58 string.

Result

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

4.1.3 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, *re-added in version 1.4.3*

- *address*

  The address as a Base58 string.

Result

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

4.1.4 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, *re-added in version 1.4.3*

- *address*

  The address as a Base58 string.

Result

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

4.1.5 blockchain.address.subscribe

Subscribe to a bitcoin address.

Signature

`blockchain.address.subscribe(address)`

Deprecated since version 1.2: removed in version 1.3, *re-added in version 1.4.3*

address

The address as a Base58 string.

Result

The *status* of the address.

Notifications

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

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

4.1.6 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 deafults 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",
```

(continues on next page)

(continued from previous page)

```
  "nonce": 3288656012,
  "prev_block_hash":
↪"000000000000000000b512b5d9fc7c5746587268547c04aa92383aaea0080289",
  "timestamp": 1520495819,
  "version": 536870912
}
```

With *raw* `true`:

```
{
  "height": 520481,
  "hex":
↪"00000020890208a0ae3a3892aa047c5468725846577cfcd9b512b5000000000000000000005dc2b02f2d297a9064ee1
↪"
}
```

Notifications

As this is a subcription, 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.

4.1.7 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 subcription, the client will receive a notification when a new block is found. The notification's signature is:

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

4.1.8 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.

4.1.9 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
}
```

4.1.10 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.

4.1.11 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"
```

CHAPTER 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.

5.1 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.

5.2 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 ot 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"
    ]
  ]
}
```

5.3 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

5.4 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()`.

5.5 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

CHAPTER 6

Peer Discovery

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

6.1 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 Fulcrum the hard-coded peers come from here: servers.json.

6.2 server.peers.subscribe

`server.peers.subscribe()` is used by SPV 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 Fulcrum, "recently" is taken to be servers it is actively able to maintain a connection to. All Fulcrum servers keep client connections to all of their peers at all times, in order to ensure that the returned list is as up-to-date as possible.

6.3 Maintaining the Peer Database

In order to keep its peer database up-to-date and fresh, Fulcrum maintains a constant connection to all of its peers, acting as an SPV client but not taking up much bandwidth. If connections are dropped or periodically fail, after some time has passed since the last successful connection to a peer, a Fulcrum server will 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.

Fulcrum 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 hour. It forgets a peer entirely if 24 hours have passed since a successful connection. Fulcrum attempts to connect to onion peers through a Tor proxy that can be configured or that it will try to autodetect.

6.4 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 SPV 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.

6.5 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.

6.6 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).

Index