Network protocol specification

Stratum platform

draft

Overview

Stratum is basically a platform for creating lightweight Bitcoin clients (= clients without a blockchain, keeping only private keys). Absence of the blockchain on the client side provide very good user experience, client has very small footprint and still provide high level of security and privacy.

More technically, Stratum is an **overlay network** on the top of Bitcoin P2P protocol, creating simplified facade for lightweight clients and hiding unnecessary complexity of decentralized protocol. However there’s much bigger potential in this overlay network than just providing simplified API for accessing blockchain stored on Stratum servers. For utilization of such potential, we definitely need some robust protocol providing enough flexibility for various type of clients and their purposes.

Some advanced ideas for Stratum network, which will need flexible network protocol:

- Integration of BTC/fiat exchanges into clients
- Wallet storages for diskless or extremely low-resource clients (AVR-based hardware wallets)
- Server-side escrows (sending bitcoins to email)
- Integration of bitcoin laundry
- Exchange calculators (for providing “fiat” equivalents of BTC in clients)
- Firstbits support
- Mining support for clients
- Various transport protocols (especially HTTP Push, which allows PHP websites to integrate with Bitcoin easily)

Requirements

1. **Protocol should be as simple as possible.** Some clients aren't capable to handle high-level message systems like Google's Protocol buffers or Apache Thrift.
2. **Protocol should be text-based.** Some languages cannot handle binary data (javascript) or it's pretty difficult to implement it correctly (PHP). It's also easier to debug text-based protocol than binary data.
3. **Protocol must support standard RPC mechanism (request-response).** Request should contains method identifier and parameters, response should be able to transfer error states/exceptions.
4. **Mapping between request and response must be clear.** Don't allow vague relation between request and response. Some message-based systems use only textual conventions to map requests and responses together, like 'firstbits_resolve <firstbits>' expects 'firstbits_response <firstbits> <address>' or 'firstbits_error <firstbits> <reason>'. It creates ambiguous data flow, avoid it.
5. **Protocol should support publish-subscribe mechanism.** Client can subscribe on server for receiving some kind of information. After this request, server will proactively broadcast messages to subscribed clients until client disconnects or cancel it's subscription.
6. **Protocol must be bi-directional.** In the opposite of standard client-server model, we sometimes need to allow server to initiate the communication. As an example, server can ask client to reconnect to another node (before switching to maintenance mode) or send some text message to user.

Network layers

Stratum network layer should be flexible on many levels. This document is proposing complete stack for Stratum networking, however such complex problem can be divided into three independent layers:

1. Transports
2. Application protocol
3. Services

Transports

Transports are the way how to obtain bi-directional communication between a lightweight client and Stratum servers. All Stratum servers should support some basic range of supported transports like plain socket transport, HTTP Poll (for environments with restrictive network access), HTTP Push (for non-interactive clients like PHP scripts), websocket/socket.io (for javascript clients) etc.

Transport is responsible for keeping the session, it's not a responsibility of any upper layer. It means that once any kind of transport is established, upper layers of protocol can send/receive messages like it's transferred over already established TCP socket.

- Socket transport
    - Standard TCP socket, clients initiate connection, then both sides can initiate transport of payload (application protocol) anytime.
- Websocket
    - It's HTTP-based, socket-like transport, widely supported by javascript clients. It's probably the best way how to provide instant notification to browser-based applications.
    - After initial handshake, it works like "Socket transport".
- HTTP Poll
    - Client performs HTTP POST every N seconds, storing payload as request body, processing server payload from response body.
    - HTTP POST without a session identifier creates new session (cookie?), client is responsible for using this cookie in following requests during the session.
    - Server store all messages for delivery to client into server-side buffer, flush them to client in following poll request.
- HTTP Push
    - Transport for clients which cannot keep open connection and they're unable to ask server frequently enough. Typical example is PHP-based website, which want to subscribe for some blockchain changes (notification about incoming transaction for given address).
    - On the beginning of the HTTP Poll session, client provide callback URL, turning

the current HTTP Poll transport into HTTP Push.
    - When there're some new data on Stratum server, server performs HTTP POST request to callback URL with payload in the request body.
    - Client must perform keep-alive HTTP Poll request for indicating it's still alive and want to receive HTTP Push messages. It can be done by crontab entry with one hour period, for example.

HTTP Poll

Important HTTP headers in the request:

- **Cookie** should contains all cookies provided by the server in previous calls, especially the STRATUM_SESSION cookie which is identifying current HTTP session. Blank or missing STRATUM_SESSION will initiate new session, even on the same TCP connection. Using many different cookies in different HTTP requests is possible, which allow serving more independent clients over one TCP connection.
- **Content-Type** must be always "application/stratum" indicating that payload is following Stratum protocol. This is especially useful for filtering requests in the server configuration (Apache/Nginx), so ports 80 and 443 can still be used for non-Stratum traffic.
- **X-Callback-Url** Provides the URL (always in the full form, including desired HTTP/HTTPS protocol type) for HTTP Push transport. Occurence of this header in the request turns the session into the HTTP Push protocol.

Important HTTP headers in the response:

- **Set-Cookie** contains cookies identifying current HTTP session and client must send back all those cookies in following request. By sending the STRATUM_SESSION cookie, server is indicating that the request/response is part of newly created session. When the client receive this cookie in the middle of the communication, he must re-initialize the session (subscribe for all services, …). This may happen when client perform request with the session which is already expired.
- **Content-Type** is always "application/stratum", indicating that response was created by Stratum application server.
- **Content-MD5** contains the MD5 hash of the body, for checking that content wasn't corrupted during the transmission.
- **Server** indicates the version of server implementation.
- **X-Session-Timeout** contains lifetime of current session (in seconds). After this time, every consequential request with the same session ID is considered as new session and client must perform initial handshake (subscribing for all services, …)
- **X-Content-SHA256** has the same purpose as Content-MD5, but provides checksum for clients which don't handle MD5. SHA256 must be presented in the clients anyway for creating Bitcoin transactions, so the possibility of checking transport checksum with SHA256 is minimizing dependencies.

Important HTTP status codes in the response:

- 200 OK - server processed the request succesfully
- 5xx Server error - client should perform consequential request with some delay to prevent server overloading.

HTTP Push

This transport works on the top of HTTP Poll transport and it is fully backward compatible. It is enabled by sending X-Callback-Url header in HTTP Poll request, indicating that server may actively send payload to the client using given URL as HTTP POST request.

Client can continue to work with the transport like it's still the HTTP Poll transport, but broadcasts from the server are delivered instantly over callback URL, without waiting on consequential poll request.

Callback URL can be changed during the session, by providing updated URL in the header. HTTP Push can be also switched back to HTTP Poll, by providing X-Callback-Url with blank string value.

Client must perform occasional HTTP polling even with HTTP Push transport, to indicate he's alive and interested in receiving callbacks. Don't forget that those keep alive calls are still valid HTTP Poll requests and server can provide real data in the response.

Keep alive poll requests must be called before session on the server timed out (timeout is provided in X-Session-Timeout response header). It's recommended to perform keep alive request few seconds before session will actually timed out to protect session before expiration thanks to temporary network failure. Don't forget that occurence of STRATUM_SESSION cookie in the keep alive response indicates expiration of previous session on the server and client must perform re-initialization all previous subscriptions.

Important HTTP headers in callback:

- Content-Type is always "application/stratum", indicating that request was created by Stratum application server.
- X-Session-Id contains session ID of HTTP Poll connection. This is useful for handling multiple connections over the same callback URL.
- X-Session-Timeout contains remaining time to session expiration in seconds.

Application protocol

Protocol itself is the way how to exchange information between client and server, using already established transport. Protocol doesn't understand the meaning of exchanged information, it only keeps the track on request-response and internal data format. Protocol format is the same for all supported transports.

Basic structure of proposed protocol is line-based, json-encoded message. Every message is on separate line and there exists only two formats of messages - **request** and **response**. Those messages use the format of JSON-RPC 2.0 protocol (http://json-rpc.org/wiki/specification).

JSON-RPC allows two types of requests. One is part of standard **RPC mechanism** when every request expects some response. Second type is the **notification** formatted as a JSON-RPC request, but it doesn't expect any kind of response. Typical example is broadcasting of new block or transaction from server to clients. The difference between RPC request and notification is that notification always has *id=null*.

Request

Every RPC request contains three parts:

- message ID - integer or string
- remote method - unicode string
- parameters - list of parameters

message ID **must** be an unique identifier of request during current transport session. It may be integer or some unique string, like UUID. ID must be unique only from one side (it means, both server and clients can initiate request with id "1"). Client or server can choose string/UUID identifier for example in the case when standard "atomic" counter isn't available (multi-processing environment like PHP servers).

Examples:

- Retrieve transaction history of given Bitcoin address (client->server):
  {"id": 1, "method": "blockchain.address.get_history", "params": ["1DiiVSnksihdpdP1Pex7jghMAZffZiBY9q"]}

- Broadcast transaction to Bitcoin network (client->server):
  {"id": 2, "method": "blockchain.transaction.broadcast", "params": ["tx_payload"]}

- Subscribe for receiving information about new blocks in blockchain (client->server):
  {"id": 3, "method": "blockchain.block.subscribe", "params": []}

- Unsubscribe from receiving information about new blocks in blockchain (client->server):
  {"id": 4, "method": "blockchain.block.unsubscribe", "params": []}

Notification

Notification is like Request, but it does not expect any response and message ID is always *null*:

- message ID - null
- remote method - unicode string
- parameters - list of parameters

Examples:

- **Broadcast message** about new block (server->client). Server don't expect any response to this message. Client must perform call “blockchain.block.subscribe” to start receiving requests like this:

{“id”: null, “method”: “blockchain.block.new”, “params”: [“block_payload”]}

Response

Every response contains following parts

- message ID - same ID as in request, for pairing request-response together
- result - any json-encoded result object (number, string, list, array, …)
- error - null or list (error code, error message)

Examples:

- Received transaction history of an address (server->client):

  {“id”: 1, result: [“tx1_id”: {}, “tx2_id”: {}], error: null}

- Received resolved firstbits address (server->client):

  {“id”: 2, result: [“1DiiVSnksihdpdP1Pex7jghMAZffZiBY9q”], “error”: null}

- Resolving of firstbits address failed (server->client):

  {“id”: 2, result: null, “error”: (1, “Firstbits cannot be translated to valid Bitcoin address”)}

Message signatures

#TODO
sign, sign_type, sign_id, sign_time

Replay attacks

#FIXME: Timestamp IS a part of signed data, it’s up to client to decide if cached message is good enough.

To protect messages against replay attacks, RPC commands vulnerable to such attack should contain some unique parameter. Because every signature is constructed from both request and response data, current timestamp in the request should provide enough security.

Timestamp is a part of signature metadata. Although some types of RPC calls are immutable by design and adding timestamp to signature doesn’t provide additional security against replay attack, it’s up to client to decide if cached response (with timestamp in the history) is good enough for him. Caching of signatures is a reasonable way to improve server load and using signature with older timestamp shouldn’t be an issue for immutable calls.

Exceptions

Every RPC error returns 3-tuple with error code, human-readable error message and detailed traceback of exception (may be just blank string on production servers for security reasons). Error codes < 0 are protocol specific or unhandled exceptions. Error codes > 0 are exceptions generated by services and every service should offer detailed description of possible error states.

-1, Unknown exception, error message should contain more specific description
-2, “Service not found”
-3, “Method not found”
-10, “Fee required”
-20, “Signature required”, when server expects request to be signed
-21, “Signature unavailable”, when server rejects to sign response
-22, “Unknown signature type”, when server doesn’t understand any signature type from “sign_type”
-23, “Bad signature”, signature doesn’t match source data

#TODO

Services

#TODO Documentation for blockchain-related services (for Electrum)

Server side:

discovery.list_services()
discovery.list_vendors(service_name)
discovery.list_methods(service_name, vendor)
discovery.list_params(service_name, vendor, method)

node.peers.subscribe()
node.stop()
node.get_signature_pubkey()

# TODO:
blockchain.block.subscribe
blockchain.block.unsubscribe
blockchain.block.broadcast

blockchain.address.subscribe
blockchain.address.unsubscribe
blockchain.address.get_history
blockchain.address.get_balance

blockchain.transaction.broadcast
blockchain.transaction.get

blockchain.transactions.guess_fee
blockchain.transactions.subscribe
blockchain.transactions.unsubscribe

firstbits.resolve
firstbits.lookup

Client side:

node.get_version
node.reconnect(hostname)

Notifications:
blockchain.block.notify()
blockchain.transaction.notify()