Stratum v2 (WORK IN PROGRESS)

```
BIP: ???
Layer: Applications
Title: Stratum mining protocol V2
Author: Pavel Moravec <pavel@braiins.com>
        Jan Čapek <jan@braiins.com>
        Matt Corallo <bipstratum@bluematt.me>
Comments-Summary: No comments yet.
Comments-URI: https://github.com/:BIP-0310
Status: Draft
Type: Informational
Created: 2019-??-??
License: BSD-3-Clause
         CC0-1.0
```

Abstract

The current stratum protocol (v1) is used prevalently throughout the cryptocurrency mining industry today, but it was never intended nor designed to be an industry standard.

This document proposes a new version of stratum protocol that addresses scaling and quality issues of the previous version, focusing on more efficient data transfers (i.e. distribution of mining jobs and result submissions) as well as increased security.

Additionally, the redesigned protocol includes support for transaction selection by the miners themselves, as opposed to the current version of the protocol in which only the pool operators can determine a new block's transaction set.

There are some trade offs necessary to make the protocol scalable and relatively simple, which will be addressed in the detailed discussions below.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

We keep the name "Stratum" so that people will recognize that this is an upgrade of the widespread protocol version 1, with the hope that it will help gather support for the new version more easily.

Motivation

Stratum protocol v1 is JSON-based and lacks cryptographic authentication, making it slower, heavier, and less secure than it can be considering the alternatives available today. Given the

cryptocurrency mining industry's continued maturation and growth, it's necessary to address v1's deficiencies and move to a more efficient solution with a precise definition.

One of the primary motivations of the new protocol is to reduce the amount and respective size of data transfers between miners, proxies, and pool operators to an absolute minimum. This will enable stratum servers to use the saved bandwidth for higher submission rates, thus yielding a reduced variance in hash rate (and in turn in miner payouts).

To increase efficiency further, we will enable a simplified mode for end mining devices which completely eliminates the need for extranonce and Merkle path handling (i.e. any coinbase modification on downstream machines). This mode, called header-only mining, makes computations simpler for miners and work validation much lighter on the server side. Furthermore, header-only mining reduces the cost of future changes to the Bitcoin protocol, as mining firmware and protocols do not need to be upgraded in conjunction with full nodes.

In terms of security, another important improvement to make is hardening the protocol against man-in-the-middle attacks by providing a way for mining devices to verify the integrity of the assigned mining jobs and other commands.

Last but not the least, this protocol strives to allow downstream nodes to choose mining jobs and efficiently communicate them to upstream nodes to reduce power currently held by mining pools (block version selection, transaction selection). This should be possible without harming public pool business models or otherwise leading to more centralization in another area of the mining industry.

Design Goals

As there are numerous changes from the original Stratum v1 to v2, it may be helpful to briefly review some high-level design goals before getting into more detailed technical specifications:

- Develop a binary protocol with a precise definition. Despite its simplicity, v1 was not a precisely defined protocol and ended up with multiple slightly different dialects. We don't want any room for different interpretations of v2.
- Make Stratum v2 logically similar to v1 whenever possible so that it's easier to understand for people who are already familiar with the protocol. V1 is widely used not only in bitcoin mining, but also for mining various altcoins.
- Remove as many issues caused by v1 as possible, based on substantial historical experience with it.
    - Remove explicit mining subscriptions (mining.subscribe) altogether. It was originally part of a more elaborate protocol and is no longer relevant.
    - Make extranonce subscription a native part of the protocol, not an extension.
    - Clean up difficulty controlling, which is really suboptimal v1.
    - Drop JSON.

    - Rework BIP310 from scratch.
- Allow different mining jobs on the same connection.
- Avoid introducing any additional risks to pool operators and miners since that would make adoption of v2 very improbable.
- Support version rolling natively. Bitcoin block header contains a version field whose bits (determined by BIP320) can be freely used to extend the hashing space for a miner. It is already a common tech, we want to include it as a first class citizen in the new protocol.
- Support header-only mining (not touching the coinbase transaction) in as many situations as possible. Header-only mining should be easier and faster on mining devices, while also decreasing network traffic.
- Dramatically reduce network traffic as well as client-side and server-side computational intensity, while still being able to send and receive hashing results rapidly for precise hash rate measurement (and therefore more precise mining reward distribution).
- Allow miners to (optionally) choose the transaction set they mine through work negotiation on some independent communication channel. At the same time, allow miners to choose how they utilize the available bits in the block header nVersion field, including both those bits which are used for mining (e.g. version-rolling AsicBoost) by BIP320, and those bits used for BIP8/9 signaling. This mechanism must not interfere with the efficiency or security of the main mining protocol.
    - Use a separate communication channel for transaction selection so that it does not have a performance impact on the main mining/share communication, as well as can be run in three modes - disabled (i.e.pool does not yet support client work selection, to provide an easier transition from Stratum v1), client-push (to maximally utilize the client's potential block-receive-latency differences from the pool), and client-negotiated (for pools worried about the potential of clients generating invalid block templates).
- Put complexity on the pool side rather than the miner side whenever possible. Keep the protocol part to be implemented in embedded devices as small and easy as possible. Mining devices tend to be difficult to update. Any mistake in a firmware can be very costly. Either on miners side (non-functioning firmware) or pool side (necessity to implement various workarounds and hacks to support misbehaving firmware).
- Allow for translation to and from the original protocol on a proxy level (e.g. different downstream devices) without the necessity to reconnect.
- Reduce the stale ratio as much as possible through efficiency improvements.
- Support/allow for nTime rolling in hardware in a safe and controlled way.
- Simple support for vendor-specific extensions without polluting the protocol, or complicating pool implementation.

- Optional telemetry data, allowing for easy monitoring of farms, without sacrificing the privacy of miners who wish to remain private.
- Allow aggregation of connections to upstream nodes with an option to aggregate or not aggregate hash rate for target setting on those connections.
- Ensure protocol design allows for devices to implement their own swarm algorithms. Mining devices can dynamically form small groups with an elected master that is responsible for aggregating connections towards upstream endpoint(s), acting as a local proxy. Aggregating connections and running multiple channels across a single TCP connection yields a better ratio of actual payload vs TCP/IP header sizes, as the share submission messages are in the range of 20 bytes. Still, to avoid overly complicating the protocol, automated negotiation of swarm/proxy detection is left to future extensions or vendor-specific messages.

Protocol Overview

There are technically four distinct (sub)protocols needed in order to fully use all of the features proposed in this document:

1. Mining Protocol: The main protocol used for mining and the direct successor of Stratum v1. A mining device uses it to communicate with its upstream node, pool, or a proxy. A proxy uses it to communicate with a pool (or another proxy). This protocol needs to be implemented in all scenarios. For cases in which a miner or pool doesn't support transaction selection, this is the only protocol used.
2. Job Negotiation Protocol: Used by a miner (a whole mining farm) to negotiate a block template with a pool. Results of this negotiation can be re-used for all mining connections to the pool to reduce computational intensity. In other words, a single negotiation can be used by an entire mining farm or even multiple farms with hundreds of thousands of devices, making it far more efficient. This is separate to allow pools to terminate such connections on separate infrastructure from mining protocol connections (i.e. share submissions). Further, such connections have very different concerns from share submissions - work negotiation likely requires, at a minimum, some spot-checking of work validity, as well as potentially substantial rate-limiting (without the inherent rate-limiting of share difficulty).
3. Template Distribution Protocol: A similarly-framed protocol for getting information about the next block out of Bitcoin Core. Designed to replace getblocktemplate with something much more efficient and easy to implement for those implementing other parts of Stratum v2.
4. Job Distribution Protocol - Simple protocol for passing newly-negotiated work to interested nodes - either proxies or miners directly. This protocol is left to be specified in a future document, as it is often unnecessary due to the Job Negotiation role being a part of a larger Mining Protocol Proxy.

Meanwhile, there are five possible roles (types of software/hardware) for communicating with these protocols.

A. Mining Device - The actual device computing the hashes. This can be further divided into header-only mining devices and standard mining devices, though most devices will likely support both modes.

B. Pool Service - Produces jobs (for those not negotiating jobs via the Job Negotiation Protocol), validates shares, and ensures blocks found by clients are propagated through the network (though clients which have full block templates MUST also propagate blocks into the Bitcoin P2P network).

C. Mining Proxy - (optional) Sits in between Mining Device(s) and Pool Service, aggregating connections for efficiency. May optionally provide additional monitoring, receive work from a Job Negotiator and use custom work with a pool, or provide other services for a farm.

D. Job Negotiator - (optional) Receives custom block templates from a Template Provider and negotiates use of the template with the pool using the Job Negotiation Protocol. Further distributes the jobs to Mining Proxy (or Proxies) using the Job Distribution Protocol. This role will often be a built-in part of a Mining Proxy.

E. Template Provider - Generates custom block templates to be passed to the Job Negotiator for eventual mining. This is usually just a Bitcoin Core full node (or possibly some other node implementation).

The Mining Protocol is used for communication between a Mining Device and Pool Service, Mining Device and Mining Proxy, Mining Proxy and Mining Proxy, or Mining Proxy and Pool Service.

The Job Negotiation Protocol is used for communication between a Job Negotiator and Pool Service.

The Template Distribution Protocol is used for communication between a Job Negotiator and Template Provider.

The Job Distribution Protocol is used for communication between a Job Negotiator and a Mining Proxy.

One type of software/hardware can fulfill more than one role (e.g. a Mining Proxy is often both a Mining Proxy and a Job Negotiator and may occasionally further contain a Template Provider in the form of a full node on the same device).

Each sub-protocol is based on the same technical principles and requires a connection oriented transport layer, such as TCP. In specific use cases, it may make sense to operate the protocol over a connectionless transport with FEC or local broadcast with retransmission. However, that is outside of the scope of this document. The minimum requirement of the transport layer is to guarantee ordered delivery of the protocol messages.

Data Types Mapping

Message definitions use common data types described here for convenience. Multibyte data types are always serialized as little-endian.

Protocol Type	Byte Length	Description
BOOL	1	Boolean value. Encoded as an unsigned 1-bit integer, True = 1, False = 0 with 7 additional padding bits in the high positions. x Recipients MUST NOT interpret bits outside of the least significant bit. Senders MAY set bits outside of the least significant bit to any value without any impact on meaning. This allows future use of other bits as flag bits.
U8	1	Unsigned integer, 8-bit
U16	2	Unsigned integer, 16-bit, little-endian
U24	3	Unsigned integer, 24-bit, little-endian (commonly deserialized as a 32-bit little-endian integer with a trailing implicit most-significant 0-byte).
U32	4	Unsigned integer, 32-bit, little-endian
U256	32	Unsigned integer, 256-bit, little-endian. Often the raw byte output of SHA-256 interpreted as an unsigned integer.
STR0_255	1 + LENGTH	1-byte length L, unsigned integer 8-bits, followed by a series of L bytes. Allowed range of length is 0 to 255. The string is not null-terminated.
B0_255	1 + LENGTH	1-byte length L, unsigned integer 8-bits, followed by a sequence of L bytes. Allowed range of length is 0 to 255.
B0_64K	2 + LENGTH	2-byte length L, unsigned little-endian integer 16-bits, followed by a sequence of L bytes. Allowed range of length is 0 to 65535.
B0_16M	3 + LENGTH	3-byte length L, encoded as a U24 above, followed by a sequence of L bytes. Allowed range of length is 0 to 2^24-1.
BYTES	LENGTH	Arbitrary sequence of LENGTH bytes. See description for how to calculate LENGTH.
PUBKEY	32	Ed25519 public key

SIGNATURE	64	Ed25519 signature
SEQ0_255[T]	Fixed size T: 1 + LENGTH * size(T) Variable length T: 1 + seq.map(|x| x.length).sum()	1-byte length L, unsigned integer 8-bits, followed by a sequence of L elements of type T. Allowed range of length is 0 to 255.
SEQ0_64K[T]	Fixed size T: 2 + LENGTH * size(T) Variable length T: 2 + seq.map(|x| x.length).sum()	2-byte length L, unsigned little-endian integer 16-bits, followed by a sequence of L elements of type T. Allowed range of length is 0 to 65535.

Framing

The protocol is binary, with fixed message framing. Each message begins with the extension type, message type, and message length (six bytes in total), followed by a variable length message. The message framing is outlined below:

Field Name	Data Type	Description
extension_type	U16	Unique identifier of the extension describing this protocol message. Most significant bit (i.e.bit 15, 0-indexed, aka **channel_msg**) indicates a message which is specific to a channel, whereas if the most significant bit is unset, the message is to be interpreted by the immediate receiving device. Note that the channel_msg bit is ignored in the extension lookup, i.e.an extension_type of 0x8ABC is for the same “extension” as 0x0ABC. If the channel_msg bit is set, the first four bytes of the payload field is a U32 representing the channel_id this message is destined for (these bytes are repeated in the message framing descriptions below). Note that for the Job Negotiation and Template Distribution Protocols the channel_msg bit is always unset.
msg_type	U8	Unique identifier of the message within the extension_type namespace.
msg_length	U24	Length of the protocol message, not including this header.
payload	BYTES	Message-specific payload of length msg_length. If the MSB in extension_type (the channel_msg bit) is set the first four bytes are defined as a U32 “channel_id”, though this definition is repeated in the message definitions below and these 4 bytes are included in msg_length.

Protocol Security

Stratum V2 employs a type of encryption scheme called AEAD (authenticated encryption with associated data) to address the security aspects of all communication that occurs between

clients and servers. This provides both confidentiality and integrity for the ciphertexts (i.e. encrypted data) being transferred, as well as providing integrity for associated data which is not encrypted. Prior to opening any Stratum V2 channels for mining, clients MUST first initiate the cryptographic session state that is used to encrypt all messages sent between themselves and servers. Thus, the cryptographic session state is independent of V2 messaging conventions.

At the same time, this specification proposes optional use of a particular handshake protocol based on the **Noise Protocol framework**. The client and server establish secure communication using Diffie-Hellman (DH) key agreement, as described in greater detail in the Authenticated Key Agreement Handshake section below.

Using the handshake protocol to establish secured communication is **optional** on the local network (e.g. local mining devices talking to a local mining proxy). However, it is **mandatory** for remote access to the upstream nodes, whether they be pool mining services, job negotiating services or template distributors.

Motivation for Authenticated Encryption with Associated Data

Data transferred by the mining protocol MUST not provide adversary information that they can use to estimate the performance of any particular miner. Any intelligence about submitted shares can be directly converted to estimations of a miner's earnings and can be associated with a particular username. This is unacceptable privacy leakage that needs to be addressed.

Motivation for Using the Noise Protocol Framework

The reasons why Noise Protocol Framework has been chosen are listed below:

- The Framework pushes to use new, modern cryptography.
- The Framework provides a formalism to describe the handshake protocol that can be verified.
- There is no legacy overhead.
- It is difficult to get wrong.
- Noise Explorer provides code generators for popular programming languages (e.g. Go, Rust).
- We can specify no flexibility (i.e. fewer degrees of freedom), helping ensure standardization of the supported ciphersuite(s).
- A custom certificate scheme is now possible (no need to use x509 certificates).

Authenticated Key Agreement Handshake

The handshake chosen for the authenticated key exchange is **Noise_NX** as it provides authentication of the server side and doesn't require authentication of the initiator (client). Server authentication is achieved implicitly via a series of Elliptic-Curve Diffie-Hellman (ECDH) operations followed by a MAC check.

The authenticated key agreement (`Noise_NX`) is performed in two distinct steps (acts). The protocol allows for secure authentication. During each act of the handshake the following occurs: some (possibly encrypted) keying material is sent to the other party; an ECDH is

performed, based on exactly which act is being executed, with the result mixed into the current set of encryption keys (`ck` the chaining key and `k` the encryption key); and an AEAD payload with a zero-length cipher text is sent. As this payload has no length, only a MAC is sent across. The mixing of ECDH outputs into a hash digest forms an incremental DoubleDH handshake.

Using the language of the Noise Protocol, **`e`** and **`s`** (both public keys with **`e`** being the **ephemeral key** and **`s`** being the **static key**) indicate possibly encrypted keying material, and **`es`**, **`ee`**, and **`se`** each indicate an ECDH operation between two keys. The handshake is laid out as follows:

```
Noise_NX(s, rs):

    -> e
    <- e, ee, s, es, SIGNATURE_NOISE_MESSAGE
```

The second handshake message is followed by a SIGNATURE_NOISE_MESSAGE. Using this additional message allows us to authenticate the stratum server to the downstream node. The certificate implements a simple 2 level public key infrastructure.

The main idea is that each stratum server is equipped with a certificate (that confirms its identity by providing signature of its "**static public key**" aka "**s**"). The certificate has time limited validity and is signed by the central pool authority.

Signature Noise Message

This message uses the same serialization format as other stratum messages. It contains serialized:

- Server Certificate header (version, valid_from and not_not_valid_after fields)
- ED25519 signature that can be verified by the Pool Authority Public key

Field Name	Data Type	Description
version	U16	Version of the certificate format
valid_from	U32	Validity start time (unix timestamp)
not_valid_after	U32	Signature is invalid after this point in time (unix timestamp)
signature	SIGNATURE	Ed25519 Signature

the client can reconstruct the full Certificate from its "**s**" and this header and authenticate the server.

Certificate format

Stratum server certificates have the following layout. The signature is constructed over the fields marked for signing after serialization using Stratum protocol binary serialization format.

Field Name	Data Type	Description	Signed Field
version	U16	Version of the certificate format	YES
valid_from	U32	Validity start time (unix timestamp)	YES
not_valid_after	U32	Signature is invalid after this point in time (unix timestamp)	YES
public_key	PUBKEY	static public key of the client	YES
authority_public_key	PUBKEY	public key used for verification of the signature	NO
signature	SIGNATURE	Ed25519 Signature	NO

URL scheme and Pool Authority Key

Downstream nodes that want to use the above outlined security scheme need to have configured the **Pool Authority Key** of the pool that they intend to connect to. The key can be embedded into the mining URL as part of the path. E.g.:

stratum2+tcp://thepool.com/u95GEReVMjK6k5YqiSFNqqTnKU4ypU2Wm8awa6tmbmDmk1bWt

The "**u95GEReVMjK6k5YqiSFNqqTnKU4ypU2Wm8awa6tmbmDmk1bWt**" is the public key in **base58-check** encoding. It is provided by the target pool and communicated to its users via a trusted channel. At least, it can be published on the pool's public website.

Reconnecting Downstream Nodes

An upstream stratum node may occasionally request reconnection of its downstream peers to a different host (e.g. due to maintenance reasons, etc.). This request is per upstream connection and affects all open channels towards the upstream stratum node.

After receiving a request to reconnect, the downstream node MUST run the handshake protocol with the new node as long as its previous connection was also running through a secure cryptographic session state.

Protocol Extensions

Protocol extensions may be defined by using a non-0 extension_type field in the message header (not including the channel_msg bit). The value used MUST either be in the range 0x4000 - 0x7fff (inclusive, i.e. have the second-to-most-significant-bit set) denoting an "experimental" extension and not be present in production equipment, or have been allocated for the purpose at http://stratumprotocol.org. While extensions SHOULD have BIPs written describing their full functionality, extension_type allocations MAY also be requested for vendor-specific proprietary extensions to be used in production hardware. This is done by sending an email with a brief description of the intended use case to the Bitcoin Protocol Development List and extensions@stratumprotocol.org. (Note that these contacts may change in the future, please check the latest version of this BIP prior to sending such a request.)

Extensions are left largely undefined in this BIP, however, there are some basic requirements that all extensions must comply with/be aware of.

For unknown extension_type's, the channel_msg bit in the extension_type field determines which device the message is intended to be processed on: if set, the channel endpoint (i.e. either an end mining device, or a pool server) is the final recipient of the message, whereas if unset, the final recipient is the endpoint of the connection on which the message is sent. Note that in cases where channels are aggregated across multiple devices, the proxy which is aggregating multiple devices into one channel forms the channel's "endpoint" and processes channel messages. Thus, any proxy devices which receive a message with the channel_msg bit set and an unknown extension_type value MUST forward that message to the downstream/upstream device which corresponds with the channel_id specified in the first four bytes of the message payload. Any channel_id mapping/conversion required for other channel messages MUST be done on the channel_id in the first four bytes of the message payload, but the message MUST NOT be otherwise modified. If a device is aware of the semantics of a given extension type, it MUST process messages for that extension in accordance with the specification for that extension.

Messages with an unknown extension_type which are to be processed locally (as defined above) MUST be discarded and ignored.

Extensions MUST require version negotiation with the recipient of the message to check that the extension is supported before sending non-version-negotiation messages for it. This prevents the needlessly wasted bandwidth and potentially serious performance degradation of extension messages when the recipient does not support them.

See ChannelEndpointChanged message in Common Protocol Messages for details about how extensions interact with dynamic channel reconfiguration in proxies.

Error Codes

The protocol uses string error codes. The list of error codes can differ between implementations, and thus implementations MUST NOT take any automated action(s) on the basis of an error code. Implementations/pools SHOULD provide documentation on the

meaning of error codes and error codes SHOULD use printable ASCII where possible. Furthermore, error codes MUST NOT include control characters.

To make interoperability simpler, the following error codes are provided which implementations SHOULD consider using for the given scenarios. Individual error codes are also specified along with their respective error messages.

- ‘unknown-user’
- ‘too-low-difficulty’
- ‘stale-share’
- ‘unsupported-feature-flags’
- ‘unsupported-protocol’
- ‘protocol-version-mismatch’

Common Protocol Messages

The following protocol messages are common across all of the protocols described in this BIP.

SetupConnection (Client -> Server)

Initiates the connection. This MUST be the first message sent by the client on the newly opened connection. Server MUST respond with either a *SetupConnection.Success* or *SetupConnection.Error* message. Clients that are not configured to provide telemetry data to the upstream node SHOULD set device_id to 0-length strings. However, they MUST always set vendor to a string describing the manufacturer/developer and firmware version and SHOULD always set hardware_version to a string describing, at least, the particular hardware/software package in use.

Field Name	Data Type	Description
protocol	U8	0 = Mining Protocol 1 = Job Negotiation Protocol 2 = Template Distribution Protocol 3 = Job Distribution Protocol
min_version	U16	The minimum protocol version the client supports (currently must be 2).
max_version	U16	The maximum protocol version the client supports (currently must be 2).
flags	U32	Flags indicating optional protocol features the client supports. Each protocol from *protocol* field has its own values/flags.
endpoint_host	STR0_255	ASCII text indicating the hostname or IP address.
endpoint_port	U16	Connecting port value.
Device information		
vendor	STR0_255	E.g. “Bitmain”

hardware_version	STR0_255	E.g. “S9i 13.5”
firmware	STR0_255	E.g. “braiins-os-2018-09-22-1-hash”
device_id	STR0_255	Unique identifier of the device as defined by the vendor.

SetupConnection.Success (Server -> Client)

Response to *SetupConnection* message if the server accepts the connection. The client is required to verify the set of feature flags that the server supports and act accordingly.

Field Name	Data Type	Description
used_version	U16	Selected version proposed by the connecting node that the upstream node supports. This version will be used on the connection for the rest of its life.
flags	U32	Flags indicating optional protocol features the server supports. Each protocol from *protocol* field has its own values/flags.

SetupConnection.Error (Server -> Client)

When protocol version negotiation fails (or there is another reason why the upstream node cannot setup the connection) the server sends this message with a particular error code prior to closing the connection.

In order to allow a client to determine the set of available features for a given server (e.g. for proxies which dynamically switch between different pools and need to be aware of supported options), clients SHOULD send a SetupConnection message with all flags set and examine the (potentially) resulting SetupConnection.Error message’s flags field. The Server MUST provide the full set of flags which it does not support in each SetupConnection.Error message and MUST consistently support the same set of flags across all servers on the same hostname and port number. If flags is 0, the error is a result of some condition aside from unsupported flags.

Field Name	Data Type	Description
flags	U32	Flags indicating features causing an error.
error_code	STR0_255	Human-readable error code(s). See Error Codes section, below.

Possible error codes:

- ‘unsupported-feature-flags’
- ‘unsupported-protocol’
- ‘protocol-version-mismatch’

ChannelEndpointChanged (Server -> Client)

When a channel's upstream or downstream endpoint changes and that channel had previously sent messages with **channel_msg** bitset of unknown extension_type, the intermediate proxy MUST send a **ChannelEndpointChanged** message. Upon receipt thereof, any extension state (including version negotiation and the presence of support for a given extension) MUST be reset and version/presence negotiation must begin again.

Field Name	Data Type	Description
channel_id	U32	The channel which has changed endpoint.

Mining Protocol

Channels

The protocol is designed such that downstream devices (or proxies) open communication channels with upstream stratum nodes within established connections. The upstream stratum endpoints could be actual mining servers or proxies that pass the messages further upstream. Each channel identifies a dedicated mining session associated with an authorized user. Upstream stratum nodes accept work submissions and specify a mining target on a per-channel basis.

There can theoretically be up to 2^{32} open channels within one physical connection to an upstream stratum node. All channels are independent of each other, but share some messages broadcasted from the server for higher efficiency (e.g. information about a new prevhash). Each channel is identified by its channel_id (U32), which is consistent throughout the whole life of the connection.

A proxy can either transparently allow its clients to open separate channels with the server (preferred behaviour) or aggregate open connections from downstream devices into its own open channel with the server and translate the messages accordingly (present mainly for allowing v1 proxies). Both options have some practical use cases. In either case, proxies SHOULD aggregate clients' channels into a smaller number of TCP connections. This saves network traffic for broadcast messages sent by a server because fewer messages need to be sent in total, which leads to lower latencies as a result. And it further increases efficiency by allowing larger packets to be sent.

The protocol defines three types of channels: **standard channels**, **extended channels** (mining sessions) and **group channels** (organizational), which are useful for different purposes.

The main difference between standard and extended channels is that standard channels cannot manipulate the coinbase transaction / Merkle path, as they operate solely on provided Merkle roots. We call this **header-only mining**. Extended channels, on the other hand, are given extensive control over the search space so that they can implement various advanced

use cases such as translation between v1 and v2 protocols, difficulty aggregation, custom search space splitting, etc.

This separation vastly simplifies the protocol implementation for clients that don't support extended channels, as they only need to implement the subset of protocol messages related to standard channels (see Mining Protocol Messages for details).

Standard Channels

Standard channels are intended to be used by end mining devices.

The size of the search space for one standard channel (header-only mining) for one particular value in the nTime field is 2^(NONCE_BITS + VERSION_ROLLING_BITS) = ~280Th, where NONCE_BITS = 32 and VERSION_ROLLING_BITS = 16. This is a guaranteed space before nTime rolling (or changing the Merkle root).

The protocol dedicates all directly modifiable bits (version, nonce, and nTime) from the block header to one mining channel. This is the smallest assignable unit of search space by the protocol. The client which opened the particular channel owns the whole assigned space and can split it further if necessary (e.g. for multiple hashing boards and for individual chips etc.).

Extended channels

Extended channels are intended to be used by proxies. Upstream servers which accept connections and provide work MUST support extended channels. Clients, on the other hand, do not have to support extended channels, as they MAY be implemented more simply with only standard channels at the end-device level. Thus, upstream servers providing work MUST also support standard channels.

The size of search space for an extended channel is 2^(NONCE_BITS+VERSION_ROLLING_BITS+extranonce_size*8) per nTime value.

Group Channels

Standard channels opened within one particular connection can be grouped together to be addressable by a common communication group channel.

Whenever a standard channel is created it is always put into some channel group identified by its group_channel_id. Group channel ID namespace is the same as channel ID namespace on a particular connection but the values chosen for group channel IDs must be distinct.

Future Jobs

An empty future block job or speculated non-empty job can be sent in advance to speedup new mining job distribution. The point is that the mining server MAY have precomputed such a job and is able to pre-distribute it for all active channels. The only missing information to start to mine on the new block is the new prevhash. This information can be provided independently.

Such an approach improves the efficiency of the protocol where the upstream node doesn't waste precious time immediately after a new block is found in the network.

Hashing Space Distribution

Each mining device has to work on a unique part of the whole search space. The full search space is defined in part by valid values in the following block header fields:

- Nonce header field (32 bits),
- Version header field (16 bits, as specified by BIP 320),
- Timestamp header field.

The other portion of the block header that's used to define the full search space is the Merkle root hash of all transactions in the block, projected to the last variable field in the block header:

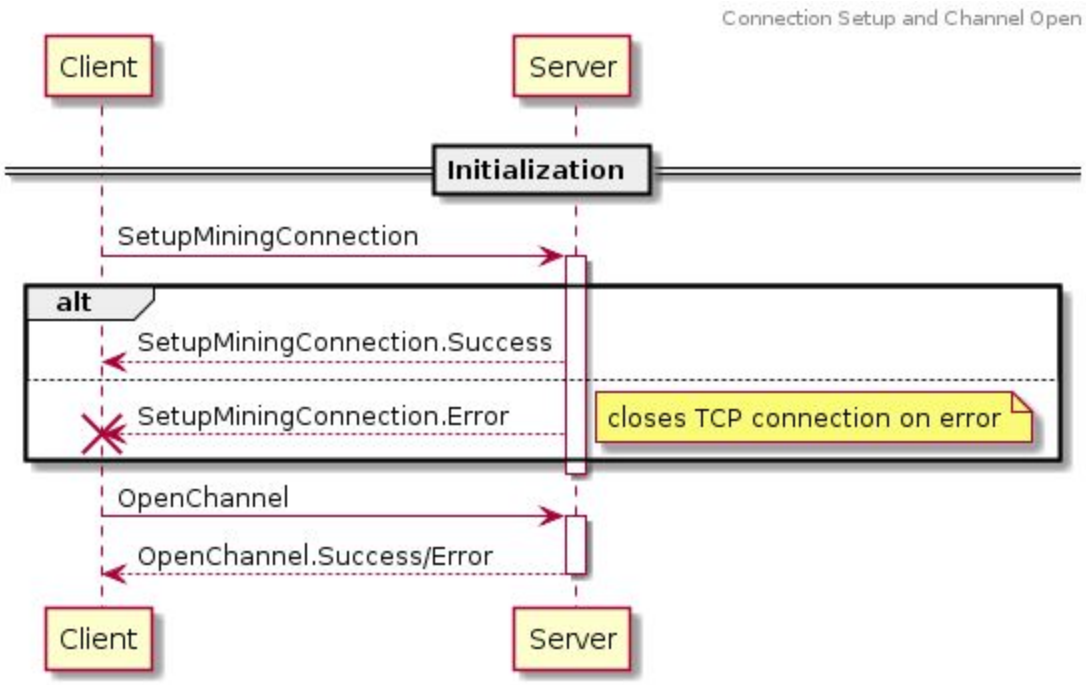
- Merkle root, deterministically computed from:
    - Coinbase transaction: typically 4-8 bytes, possibly much more.
    - Transaction set: practically unbounded space. All roles in Stratum v2 MUST NOT use transaction selection/ordering for additional hash space extension. This stems both from the concept that miners/pools should be able to choose their transaction set freely without any interference with the protocol, and also to enable future protocol modifications to Bitcoin. In other words, any rules imposed on transaction selection/ordering by miners not described in the rest of this document may result in invalid work/blocks.

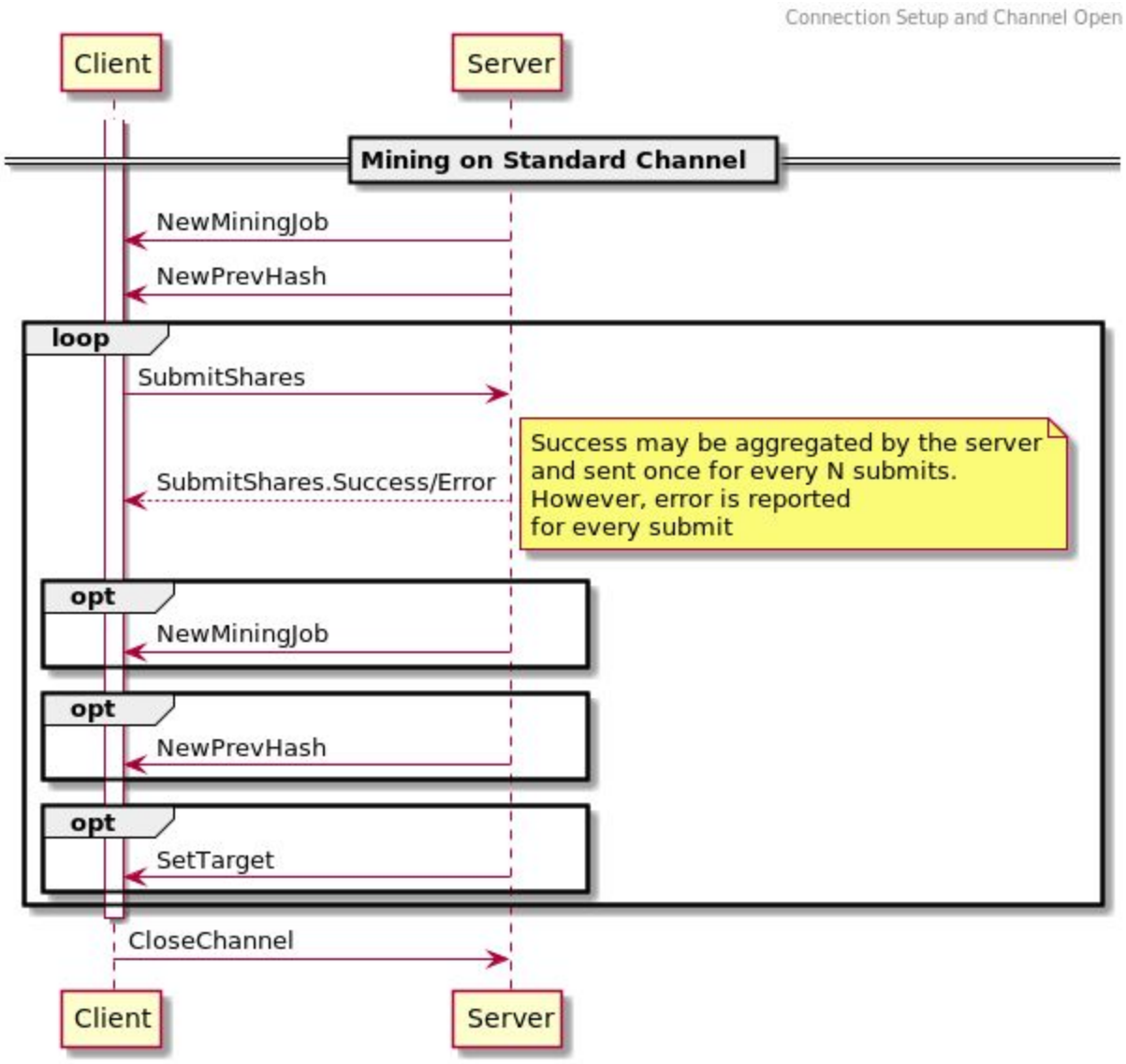
Mining servers MUST assign a unique subset of the search space to each connection/channel (and therefore each mining device) frequently and rapidly enough so that the mining devices are not running out of search space. Unique jobs can be generated regularly by:

- Putting unique data into the coinbase for each connection/channel, and/or
- Using unique work from a work provider, e.g. a previous work update (note that this is likely more difficult to implement, especially in light of the requirement that transaction selection/ordering not be used explicitly for additional hash space distribution).

This protocol explicitly expects that upstream server software is able to manage the size of the hashing space correctly for its clients and can provide new jobs quickly enough.

Mining Protocol Messages





SetupConnection flags for Mining Protocol

Flags usable in SetupConnection.flags and SetupConnection.Error::flags:

Field Name	Bit	Description
REQUIRES_STANDARD_JOBS	0	The downstream node requires standard jobs. It doesn't understand group channels - it is unable to process extended jobs sent to standard channels through a group channel.
REQUIRES_WORK_SELECTION	1	If set to 1, the client notifies the server that it will send SetCustomMiningJob on this connection.
REQUIRES_VERSION_ROLLING	2	The client requires version rolling for efficiency or correct

		operation and the server MUST NOT send jobs which do not allow version rolling.

SetupConnection.Success.flags:

Field Name	Bit	Description
REQUIRES_FIXED_VERSION	0	Upstream node will not accept any changes to the version field. Note that if REQUIRES_VERSION_ROLLING was set in the SetupConnection::flags field, this bit MUST NOT be set. Further, if this bit is set, extended jobs MUST NOT indicate support for version rolling.
REQUIRES_EXTENDED_CHANNELS	1	Upstream node will not accept opening of a standard channel.

OpenStandardMiningChannel (Client -> Server)

This message requests to open a standard channel to the upstream node.

After receiving a SetupConnection.Success message, the client SHOULD respond by opening channels on the connection. If no channels are opened within a reasonable period the server SHOULD close the connection for inactivity.

Every client SHOULD start its communication with an upstream node by opening a channel, which is necessary for almost all later communication. The upstream node either passes opening the channel further or has enough local information to handle channel opening on its own (this is mainly intended for v1 proxies).

Clients must also communicate information about their hashing power in order to receive well-calibrated job assignments.

Field Name	Data Type	Description
request_id	U32	Client-specified identifier for matching responses from upstream server. The value MUST be connection-wide unique and is not interpreted by the server.
user_identity	STR0_255	Unconstrained sequence of bytes. Whatever is needed by upstream node to identify/authenticate the client, e.g. “braiinstest.worker1”. Additional restrictions can be imposed by the upstream node (e.g. a pool). It is highly recommended that UTF-8 encoding is used.
nominal_hash_rate	F32	[h/s] Expected hash rate of the device (or cumulative hashrate on the channel if multiple devices are connected downstream) in h/s. Depending on server’s target setting policy, this value can be used for

		setting a reasonable target for the channel. Proxy MUST send 0.0f when there are no mining devices connected yet.
max_target	U256	Maximum target which can be accepted by the connected device or devices. Server MUST accept the target or respond by sending OpenMiningChannel.Error message.

OpenStandardMiningChannel.Success (Server -> Client)

Sent as a response for opening a standard channel, if successful.

Field Name	Data Type	Description
request_id	U32	Client-specified request ID from OpenStandardMiningChannel message, so that the client can pair responses with open channel requests.
channel_id	U32	Newly assigned identifier of the channel, stable for the whole lifetime of the connection. E.g. it is used for broadcasting new jobs by NewExtendedMiningJob.
target	U256	Initial target for the mining channel.
extranonce_prefix	B0_32	Bytes used as implicit first part of extranonce for the scenario when extended job is served by the upstream node for a set of standard channels that belong to the same group.
group_channel_id	U32	Group channel into which the new channel belongs. See SetGroupChannel for details.

OpenExtendedMiningChannel (Client -> Server)

Similar to *OpenStandardMiningChannel* but requests to open an extended channel instead of standard channel.

Field Name	Data Type	Description
<All fields from *OpenStandardMiningChannel*>		
min_extranonce_size	U16	Minimum size of extranonce needed by the device/node.

OpenExtendedMiningChannel.Success (Server -> Client)

Sent as a response for opening an extended channel.

Field Name	Data Type	Description
request_id	U32	Client-specified request ID from OpenExtendedMiningChannel message, so that the client can pair responses with open channel requests.
channel_id	U32	Newly assigned identifier of the channel, stable for the whole lifetime of the connection. E.g. it is used for broadcasting new jobs by NewExtendedMiningJob.
target	U256	Initial target for the mining channel.
extranonce_size	U16	Extranonce size (in bytes) set for the channel.
extranonce_prefix	B0_32	Bytes used as implicit first part of extranonce.

OpenMiningChannel.Error (Server -> Client)

Field Name	Data Type	Description
request_id	U32	Client-specified request ID from OpenMiningChannel message.
error_code	STR0_32	Human-readable error code(s). See Error Codes section, below

Possible error codes:

- ‘unknown-user’
- ‘max-target-out-of-range’

UpdateChannel (Client -> Server)

Client notifies the server about changes on the specified channel. If a client performs device/connection aggregation (i.e. it is a proxy), it MUST send this message when downstream channels change. This update can be debounced so that it is not sent more often than once in a second (for a very busy proxy).

Field Name	Data Type	Description
channel_id	U32	Channel identification.
nominal_hash_rate	F32	See Open*Channel for details.
maximum_target	U256	Maximum target is changed by server by sending SetTarget. This field is understood as device’s request. There can be some delay between UpdateChannel and corresponding SetTarget messages, based on new job readiness on the server.

When maximum_target is smaller than currently used maximum target for the channel, upstream node MUST reflect the client’s request (and send appropriate SetTarget message).

UpdateChannel.Error (Server -> Client)

Sent only when UpdateChannel message is invalid. When it is accepted by the server, no response is sent back.

Field Name	Data Type	Description
channel_id	U32	Channel identification.
error_code	STR0_32	Human-readable error code(s). See Error Codes section, below

Possible error codes:

- 'max-target-out-of-range'
- 'invalid-channel-id'

CloseChannel (Client -> Server, Server -> Client)

Client sends this message when it ends its operation. The server MUST stop sending messages for the channel. A proxy MUST send this message on behalf of all opened channels from a downstream connection in case of downstream connection closure.

Field Name	Data Type	Description
channel_id	U32	Channel identification.
reason_code	STR0_32	Reason for closing the channel.

If a proxy is operating in channel aggregating mode (translating downstream channels into aggregated extended upstream channels), it MUST send an UpdateChannel message when it receives CloseChannel or connection closure from a downstream connection. In general, proxy servers MUST keep the upstream node notified about the real state of the downstream channels.

SetExtranoncePrefix (Server -> Client)

Changes downstream node's extranonce prefix. It is applicable for all jobs sent after this message on a given channel (both jobs provided by the upstream or jobs introduced by SetCustomMiningJob message). This message is applicable only for explicitly opened extended channels or standard channels (not group channels).

Field Name	Data Type	Description
channel_id	U32	Extended or standard channel identifier.
extranonce_prefix	B0_32	Bytes used as implicit first part of extranonce.

SubmitSharesStandard (Client -> Server)

Client sends result of its hashing work to the server.

Field Name	Data Type	Description
channel_id	U32	Channel identification.
sequence_number	U32	Unique sequential identifier of the submit within the channel.
job_id	U32	Identifier of the job as provided by *NewMiningJob* or *NewExtendedMiningJob* message.
nonce	U32	Nonce leading to the hash being submitted.
ntime	U32	The nTime field in the block header. This MUST be greater than or equal to the header_timestamp field in the latest SetNewPrevHash message and lower than or equal to that value plus the number of seconds since the receipt of that message.
version	U32	Full nVersion field.

SubmitSharesExtended (Client -> Server)

Only relevant for extended channels. The message is the same as SubmitShares, with the following additional field:

Field Name	Data Type	Description
<SubmitSharesStandard message fields>		
extranonce	B0_31	Extranonce bytes which need to be added to coinbase to form a fully valid submission (full coinbase = coinbase_tx_prefix + extranonce_prefix + extranonce + coinbase_tx_suffix). The size of the provided extranonce MUST be equal to the negotiated extranonce size from channel opening.

SubmitShares.Success (Server -> Client)

Response to SubmitShares or SubmitSharesExtended, accepting results from the miner. Because it is a common case that shares submission is successful, this response can be provided for multiple SubmitShare messages aggregated together.

Field Name	Data Type	Description
channel_id	U32	Channel identifier.

last_sequence_number	U32	Most recent sequence number with a correct result.
new_submits_accepted_count	U32	Count of new submits acknowledged within this batch.
new_shares_sum	U64	Sum of shares acknowledged within this batch.

The server doesn't have to double check that the sequence numbers sent by a client are actually increasing. It can simply use the last one received when sending a response. It is the client's responsibility to keep the sequence numbers correct/useful.

SubmitShares.Error (Server -> Client)

An error is immediately submitted for every incorrect submit attempt. In case the server is not able to immediately validate the submission, the error is sent as soon as the result is known. This delayed validation can occur when a miner gets faster updates about a new prevhash than the server does (see NewPrevHash message for details).

Field Name	Data Type	Description
channel_id	U32	Channel identifier.
sequence_number	U32	Submission sequence number for which this error is returned.
error_code	STR0_32	Human-readable error code(s). See Error Codes section, below

Possible error codes:

- 'invalid-channel-id'
- 'stale-share'
- 'difficulty-too-low'

NewMiningJob (Server -> Client)

The server provides an updated mining job to the client through a standard channel.

If the future_job field is set to *False*, the client MUST start to mine on the new job as soon as possible after receiving this message.

Field Name	Data Type	Description
channel_id	U32	Channel identifier, this must be a standard channel.
job_id	U32	Server's identification of the mining job. This identifier must be provided to the server when shares are submitted later in the mining process.
future_job	BOOL	True if the job is intended for a future *SetNewPrevHash* message sent on

		this channel. If False, the job relates to the last sent *SetNewPrevHash* message on the channel and the miner should start to work on the job immediately.
<Bitcoin specific part>		
version	U32	Valid version field that reflects the current network consensus. The general purpose bits (as specified in BIP320) can be freely manipulated by the downstream node. The downstream node MUST NOT rely on the upstream node to set the BIP320 bits to any particular value.
merkle_root	B32	Merkle root field as used in the bitcoin block header.

NewExtendedMiningJob (Server -> Client)

(Extended and group channels only)

For an **extended channel**: The whole search space of the job is owned by the specified channel. If the future_job field is set to *False*, the client MUST start to mine on the new job as soon as possible after receiving this message.

For a **group channel**: This is a broadcast variant of NewMiningJob message with the merkle_root field replaced by merkle_path and coinbase TX prefix and suffix, for further traffic optimization. The Merkle root is then defined deterministically for each channel by the common merkle_path and unique extranonce_prefix serialized into the coinbase. The full coinbase is then constructed as follows: *coinbase_tx_prefix* + *extranonce_prefix* + *coinbase_tx_suffix*.

The proxy MAY transform this multicast variant for downstream standard channels into NewMiningJob messages by computing the derived Merkle root for them. A proxy MUST translate the message for all downstream channels belonging to the group which don't signal that they accept extended mining jobs in the SetupConnection message (intended and expected behaviour for end mining devices).

Field Name	Data Type	Description
channel_id	U32	For a group channel, the message is broadcasted to all standard channels belonging to the group. Otherwise, it is addressed to the specified extended channel.
job_id	U32	Server's identification of the mining job.
future_job	BOOL	True if the job is intended for a future *SetNewPrevHash* message sent on the channel. If False, the job relates to the last sent *SetNewPrevHash* message on the channel and the miner should start to work on the job immediately.

version	U32	Valid version field that reflects the current network consensus.
version_rolling_allowed	BOOL	If set to True, the general purpose bits of version (as specified in BIP320) can be freely manipulated by the downstream node. The downstream node MUST NOT rely on the upstream node to set the BIP320 bits to any particular value. If set to False, the downstream node MUST use version as it is defined by this message.
merkle_path	SEQ0_255[U256]	Merkle path hashes ordered from deepest.
coinbase_tx_prefix	B0_64K	Prefix part of the coinbase transaction*.
coinbase_tx_suffix	B0_64K	Suffix part of the coinbase transaction.

*The full coinbase is constructed by inserting one of the following:

- For a **standard channel:** extranonce_prefix
- For an **extended channel**: extranonce_prefix + extranonce (=N bytes), where N is the negotiated extranonce space for the channel (OpenMiningChannel.Success.extranonce_size)

SetNewPrevHash (Server -> Client, broadcast)

Prevhash is distributed whenever a new block is detected in the network by an upstream node. This message MAY be shared by all downstream nodes (sent only once to each channel group). Clients MUST immediately start to mine on the provided prevhash. When a client receives this message, only the job referenced by Job ID is valid. The remaining jobs already queued by the client have to be made invalid.

Note: There is no need for block height in this message.

Field Name	Data Type	Description
channel_id	U32	Group channel or channel that this prevhash is valid for.
job_id	U32	ID of a job that is to be used for mining with this prevhash. A pool may have provided multiple jobs for the next block height (e.g. an empty block or a block with transactions that are complementary to the set of transactions present in the current block template).
prev_hash	U256	Previous block's hash, block header field.
min_ntime	U32	Smallest nTime value available for hashing.
nbits	U32	Block header field.

SetCustomMiningJob (Client -> Server)

Can be sent only on extended channel. SetupConnection.flags MUST contain *REQUIRES_WORK_SELECTION* flag (work selection feature successfully negotiated).

The downstream node has a custom job negotiated by a trusted external Job Negotiator. The mining_job_token provides the information for the pool to authorize the custom job that has been or will be negotiated between the Job Negotiator and Pool.

Field Name	Data Type	Description
channel_id	U32	Extended channel identifier.
request_id	U32	Client-specified identifier for pairing responses.
mining_job_token	B0_255	Token provided by the pool which uniquely identifies the job that the Job Negotiator has negotiated with the pool. See the Job Negotiation Protocol for more details.
version	U32	Valid version field that reflects the current network consensus. The general purpose bits (as specified in BIP320) can be freely manipulated by the downstream node.
prev_hash	U256	Previous block's hash, found in the block header field.
min_ntime	U32	Smallest nTime value available for hashing.
nbits	U32	Block header field.
coinbase_tx_version	U32	The coinbase transaction nVersion field.
coinbase_prefix	B0_255	Up to 8 bytes (not including the length byte) which are to be placed at the beginning of the coinbase field in the coinbase transaction.
coinbase_tx_input_nSequence	U32	The coinbase transaction input's nSequence field.
coinbase_tx_value_remaining	U64	The value, in satoshis, available for spending in coinbase outputs added by the client. Includes both transaction fees and block subsidy.
coinbase_tx_outputs	SEQ0_64K[B0_64K]	Bitcoin transaction outputs to be included as the last outputs in the coinbase transaction.
coinbase_tx_locktime	U32	The locktime field in the coinbase transaction.
merkle_path	SEQ0_255[U256]	Merkle path hashes ordered from deepest.

extranonce_size	U16	Size of extranonce in bytes that will be provided by the downstream node.
future_job	BOOL	TBD: Can be custom job ever future?

SetCustomMiningJob.Success (Server -> Client)

Response from the server when it accepts the custom mining job. Client can start to mine on the job immediately (by using the job_id provided within this response).

Field Name	Data Type	Description
channel_id	U32	Extended channel identifier.
request_id	U32	Client-specified identifier for pairing responses. Value from the request MUST be provided by upstream in the response message.
job_id	U32	Server's identification of the mining job.
coinbase_tx_prefix	B0_64K	Prefix part of the coinbase transaction*.
coinbase_tx_suffix	B0_64K	Suffix part of the coinbase transaction.

SetCustomMiningJob.Error (Server -> Client)

Field Name	Data Type	Description
channel_id	U32	Extended channel identifier.
request_id	U32	Client-specified identifier for pairing responses. Value from the request MUST be provided by upstream in the response message.
error_code	STR0_32	Reason why the custom job has been rejected.

Possible errors:

- 'invalid-channel-id'
- 'invalid-mining-job-token'
- 'invalid-job-param-value-{}' - {} is replaced by a particular field name from SetCustomMiningJob message

SetTarget (Server -> Client)

The server controls the submission rate by adjusting the difficulty target on a specified channel. All submits leading to hashes higher than the specified target will be rejected by the server.

Maximum target is valid until the next *SetTarget* message is sent and is applicable for all jobs received on the channel in the future or already received with flag `future_job=True`. The message is not applicable for already received jobs with `future_job=False`, as their maximum target remains stable.

Field Name	Data Type	Description
channel_id	U32	Channel identifier.
maximum_target	U256	Maximum value of produced hash that will be accepted by a server to accept shares.

When SetTarget is sent to a group channel, the maximum target is applicable to all channels in the group.

Reconnect (Server -> Client)

This message allows clients to be redirected to a new upstream node.

Field Name	Data Type	Description
new_host	STR0_255	When empty, downstream node attempts to reconnect to its present host.
new_port	U16	When 0, downstream node attempts to reconnect to its present port.

This message is connection-related so that it should not be propagated downstream by intermediate proxies. Upon receiving the message, the client re-initiates the Noise handshake and uses the pool's authority public key to verify that the certificate presented by the new server has a valid signature.

For security reasons, it is not possible to reconnect to a server with a certificate signed by a different pool authority key. The message intentionally does *not* contain a **pool public key** and thus cannot be used to reconnect to a different pool. This ensures that an attacker will not be able to redirect hashrate to an arbitrary server should the pool server get compromised and instructed to send reconnects to a new location.

SetGroupChannel (Server -> Client)

Every standard channel is a member of a group of standard channels, addressed by the upstream server's provided identifier. The group channel is used mainly for efficient job distribution to multiple standard channels at once.

If we want to allow different jobs to be served to different standard channels (e.g. because of different BIP 8 version bits) and still be able to distribute the work by sending NewExtendendedMiningJob instead of a repeated NewMiningJob, we need a more fine-grained grouping for standard channels.

This message associates a set of standard channels with a group channel. A channel (identified by particular ID) becomes a group channel when it is used by this message as group_channel_id. The server MUST ensure that a group channel has a unique channel ID within one connection. Channel reinterpretation is not allowed.

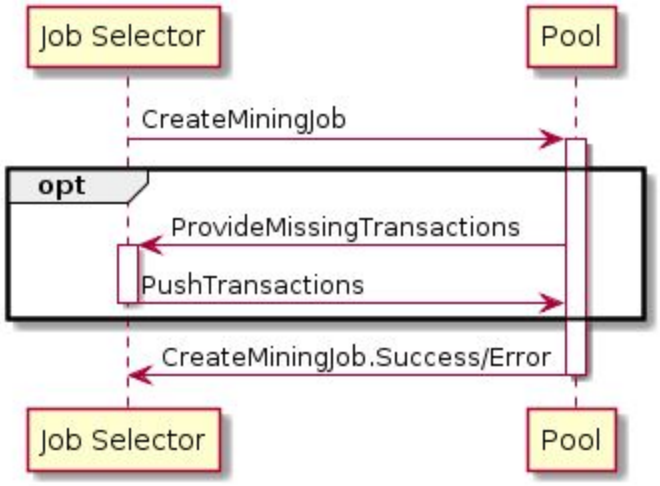
This message can be sent only to connections that don't have REQUIRES_STANDARD_JOBS flag in SetupConnection.

Field Name	Data Type	Description
group_channel_id	U32	Identifier of the group where the standard channel belongs.
channel_ids	SEQ0_64K[U32]	A sequence of opened standard channel IDs, for which the group channel is being redefined.

Job Negotiation Protocol

As outlined above, this protocol runs between the Job Negotiator and Pool and can be provided as a trusted 3rd party service for mining farms.

Protocol flow:



Job Negotiation Protocol Messages

SetupConnection flags for Job Negotiation Protocol

Flags usable in SetupConnection.flags and SetupConnection.Error::flags:

Field Name	Bit	Description
REQUIRES_ASYNC_JOB_MINING	0	The Job Negotiator requires that the mining_job_token in AllocateMiningJobToken.Success can be used immediately on a mining connection in SetCustomMiningJob message, even before CommitMiningJob and CommitMiningJob.Success messages have been sent and received. The server MUST only send AllocateMiningJobToken.Success messages with async_mining_allowed set.

No flags are yet defined for use in SetupConnection.Success.

AllocateMiningJobToken(Client->Server)

A request to get an identifier for a future-submitted mining job. Ratelimited to a rather slow rate and only available on connections where this has been negotiated. Otherwise, only mining_job_token(s) from CreateMiningJob.Success are valid.

Field Name	Data Type	Description
user_identifier	STR0_255	Unconstrained sequence of bytes. Whatever is needed by the pool to identify/authenticate the client, e.g. “braiinstest”. Additional restrictions can be imposed by the pool. It is highly recommended that UTF-8 encoding is used.
request_id	U32	Unique identifier for pairing the response.

AllocateMiningJobToken.Success(Server -> Client)

The Server MUST NOT change the value of coinbase_output_max_additional_size in AllocateMiningJobToken.Success messages unless required for changes to the pool's configuration. Notably, if the pool intends to change the space it requires for coinbase transaction outputs regularly, it should simply prefer to use the maximum of all such output sizes as the coinbase_output_max_additional_size value.

Field Name	Data Type	Description
request_id	U32	Unique identifier for pairing the response.
mining_job_token	B0_255	Token that makes the client eligible for committing a mining job for approval/transaction negotiation or for identifying custom mining job on mining connection.
coinbase_output_max_additional_size	U32	The maximum additional serialized bytes which the pool will add in coinbase transaction outputs. See discussion in the Template Distribution Protocol's CoinbaseOutputDataSize message for more details.
async_mining_allowed	BOOL	If true, the mining_job_token can be used immediately on a mining connection in the SetCustomMiningJob message, even before CommitMiningJob and CommitMiningJob.Success messages have been sent and received. If false, Job Negotiator MUST use this token for CommitMiningJob only. This MUST be true when SetupConnection.flags had REQUIRES_ASYNC_JOB_MINING set.

CommitMiningJob (Client -> Server)

A request sent by the Job Negotiator that proposes a selected set of transactions to the upstream (pool) node.

Field Name	Data Type	Description
request_id	U32	Unique identifier for pairing the response.
mining_job_token	B0_255	Previously reserved mining job token received by AllocateMiningJobToken.Success.
version	U32	Version header field. To be later modified by BIP320-consistent changes.
coinbase_tx_version	U32	The coinbase transaction nVersion field.

coinbase_prefix	B0_255	Up to 8 bytes (not including the length byte) which are to be placed at the beginning of the coinbase field in the coinbase transaction.
coinbase_tx_input_nSequence	U32	The coinbase transaction input's nSequence field.
coinbase_tx_value_remaining	U64	The value, in satoshis, available for spending in coinbase outputs added by the client. Includes both transaction fees and block subsidy.
coinbase_tx_outputs	SEQ0_64K[B0_64K]	Bitcoin transaction outputs to be included as the last outputs in the coinbase transaction.
coinbase_tx_locktime	U32	The locktime field in the coinbase transaction.
min_extranonce_size	U16	Extranonce size requested to be always available for the mining channel when this job is used on a mining connection.
tx_short_hash_nonce	U64	A unique nonce used to ensure tx_short_hash collisions are uncorrelated across the network.
tx_short_hash_list	SEQ0_64K[B8]	Sequence of SipHash-2-4(SHA256(transaction_data), tx_short_hash_nonce)) upstream node to check against its mempool. Does not include the coinbase transaction (as there is no corresponding full data for it yet).
tx_hash_list_hash	U256	Hash of the full sequence of SHA256(transaction_data) contained in the transaction_hash_list.
excess_data	B0_64K	Extra data which the Pool may require to validate the work (as defined in the Template Distribution Protocol).

CommitMiningJob.Success (Server->Client)

Field Name	Data Type	Description
request_id	U32	Identifier of the original request.
new_mining_job_token	B0_255	Unique identifier provided by the pool of the job that the Job Negotiator has negotiated with the pool. It MAY be the same token as CommitMiningJob::mining_job_token if the pool allows to start mining on not yet negotiated job. If the token is different from the one in the corresponding CommitMiningJob message (irrespective of if the client is already mining using the original token), the client MUST send a SetCustomMiningJob message on each Mining Protocol client which wishes to mine using the negotiated job.

CommitMiningJob.Error (Server->Client)

Field Name	Data Type	Description
request_id	U32	Identifier of the original request.
error_code	STR0_255	
error_details	B0_64K	Optional data providing further details to given error.

Possible error codes:

- 'invalid-mining-job-token'
- 'invalid-job-param-value-{}' - {} is replaced by a particular field name from CommitMiningJob message

IdentifyTransactions (Server->Client)

Sent by the Server in response to a CommitMiningJob message indicating it detected a collision in the tx_short_hash_list, or was unable to reconstruct the tx_hash_list_hash.

Field Name	Data Type	Description
request_id	U32	Unique identifier for pairing the response to the CommitMiningJob message.

IdentifyTransactions.Success (Client->Server)

Sent by the Client in response to an IdentifyTransactions message to provide the full set of transaction data hashes.

Field Name	Data Type	Description
request_id	U32	Unique identifier for pairing the response to the CommitMiningJob/IdentifyTransactions message.
tx_hash_list	SEQ0_64K[U256]	The full list of transaction data hashes used to build the mining job in the corresponding CommitMiningJob message.

ProvideMissingTransactions (Server->Client)

Field Name	Data Type	Description
request_id	U32	Identifier of the original CreateMiningJob request.
unknown_tx_position_list	SEQ0_64K[U16]	A list of unrecognized transactions that need to be supplied by the Job Negotiator in full. They are specified by their position in

		the original CommitMiningJob message, 0-indexed not including the coinbase transaction.

ProvideMissingTransactions.Success (Client->Server)

This is a message to push transactions that the server didn't recognize and requested them to be supplied in ProvideMissingTransactions.

Field Name	Data Type	Description
request_id	U32	Identifier of the original CreateMiningJob request.
transaction_list	SEQ0_64K[B0_16M]	List of full transactions as requested by ProvideMissingTransactions, in the order they were requested in ProvideMissingTransactions.

Template Distribution Protocol

The Template Distribution protocol is used to receive updates of the block template to use in mining the next block. It effectively replaces BIPs 22 and 23 (getblocktemplate) and provides a much more efficient API which allows Bitcoin Core (or some other full node software) to push template updates at more appropriate times as well as provide a template which may be mined on quickly for the block-after-next. While not recommended, the template update protocol can be a remote server, and is thus authenticated and signed in the same way as all other protocols (using the same SetupConnection handshake).

Like the Job Negotiation and Job Distribution protocols, all Template Distribution messages have the channel_msg bit unset, and there is no concept of channels. After the initial common handshake, the client MUST immediately send a CoinbaseOutputDataSize message to indicate the space it requires for coinbase output addition, to which the server MUST immediately reply with the current best block template it has available to the client. Thereafter, the server SHOULD push new block templates to the client whenever the total fee in the current block template increases materially, and MUST send updated block templates whenever it learns of a new block.

Template Providers MUST attempt to broadcast blocks which are mined using work they provided, and thus MUST track the work which they provided to clients.

CoinbaseOutputDataSize (Client -> Server)

Ultimately, the pool is responsible for adding coinbase transaction outputs for payouts and other uses, and thus the Template Provider will need to consider this additional block size when selecting transactions for inclusion in a block (to not create an invalid, oversized block).

Thus, this message is used to indicate that some additional space in the block/coinbase transaction be reserved for the pool's use (while always assuming the pool will use the entirety of available coinbase space).

The Job Negotiator MUST discover the maximum serialized size of the additional outputs which will be added by the pool(s) it intends to use this work. It then MUST communicate the maximum such size to the Template Provider via this message. The Template Provider MUST NOT provide NewWork messages which would represent consensus-invalid blocks once this additional size — along with a maximally-sized (100 byte) coinbase field — is added. Further, the Template Provider MUST consider the maximum additional bytes required in the output count variable-length integer in the coinbase transaction when complying with the size limits.

Field Name	Data Type	Description
coinbase_output_max_additional_size	U32	The maximum additional serialized bytes which the pool will add in coinbase transaction outputs.

NewTemplate (Server -> Client)

The primary template-providing function. Note that the coinbase_tx_outputs bytes will appear as is at the end of the coinbase transaction.

Field Name	Data Type	Description
template_id	U64	Server's identification of the template. Strictly increasing, the current UNIX time may be used in place of an ID.
future_template	BOOL	True if the template is intended for future *SetNewPrevHash* message sent on the channel. If False, the job relates to the last sent *SetNewPrevHash* message on the channel and the miner should start to work on the job immediately.
version	U32	Valid header version field that reflects the current network consensus. The general purpose bits (as specified in BIP320) can be freely manipulated by the downstream node. The downstream node MUST NOT rely on the upstream node to set the BIP320 bits to any particular value.
coinbase_tx_version	U32	The coinbase transaction nVersion field.
coinbase_prefix	B0_255	Up to 8 bytes (not including the length byte) which are to be placed at the beginning of the coinbase field in the coinbase transaction.
coinbase_tx_input_sequence	U32	The coinbase transaction input's nSequence field.
coinbase_tx_value_remaining	U64	The value, in satoshis, available for spending in coinbase outputs added by the client. Includes both transaction fees and block subsidy.

coinbase_tx_outputs_count	U32	The number of transaction outputs included in coinbase_tx_outputs.
coinbase_tx_outputs	B0_64K	Bitcoin transaction outputs to be included as the last outputs in the coinbase transaction.
coinbase_tx_locktime	U32	The locktime field in the coinbase transaction.
merkle_path	SEQ0_255[U256]	Merkle path hashes ordered from deepest.

SetNewPrevHash (Server -> Client)

Upon successful validation of a new best block, the server MUST immediately provide a SetNewPrevHash message. If a NewWork message has previously been sent with the future_job flag set, which is valid work based on the prev_hash contained in this message, the template_id field SHOULD be set to the job_id present in that NewTemplate message indicating the client MUST begin mining on that template as soon as possible.

TODO: Define how many previous works the client has to track (2? 3?), and require that the server reference one of those in SetNewPrevHash.

Field Name	Data Type	Description
template_id	U64	template_id referenced in a previous NewTemplate message.
prev_hash	U256	Previous block's hash, as it must appear in the next block's header.
header_timestamp	U32	The nTime field in the block header at which the client should start (usually current time). This is NOT the minimum valid nTime value.
nBits	U32	Block header field.
target	U256	The maximum double-SHA256 hash value which would represent a valid block. Note that this may be lower than the target implied by nBits in several cases, including weak-block based block propagation.

RequestTransactionData (Client -> Server)

A request sent by the Job Negotiator to the Template Provider which requests the set of transaction data for all transactions (excluding the coinbase transaction) included in a block, as well as any additional data which may be required by the Pool to validate the work.

Field Name	Data Type	Description
template_id	U64	The template_id corresponding to a NewTemplate message.

RequestTransactionData.Success (Server->Client)

A response to RequestTransactionData which contains the set of full transaction data and excess data required for validation. For practical purposes, the excess data is usually the SegWit commitment, however the Job Negotiator MUST NOT parse or interpret the excess data in any way. Note that the transaction data MUST be treated as opaque blobs and MUST include any SegWit or other data which the Pool may require to verify the transaction. For practical purposes, the transaction data is likely the witness-encoded transaction today. However, to ensure backward compatibility, the transaction data MAY be encoded in a way that is different from the consensus serialization of Bitcoin transactions.

Ultimately, having some method of negotiating the specific format of transactions between the Template Provider and the Pool's Template verification node would be overly burdensome, thus the following requirements are made explicit. The RequestTransactionData.Success sender MUST ensure that the data is provided in a forwards- and backwards-compatible way to ensure the end receiver of the data can interpret it, even in the face of new, consensus-optional data. This allows significantly more flexibility on both the RequestTransactionData.Success-generating and -interpreting sides during upgrades, at the cost of breaking some potential optimizations which would require version negotiation to provide support for previous versions. For practical purposes, and as a non-normative suggested implementation for Bitcoin Core, this implies that additional consensus-optional data be appended at the end of transaction data. It will simply be ignored by versions which do not understand it.

To work around the limitation of not being able to negotiate e.g. a transaction compression scheme, the format of the opaque data in RequestTransactionData.Success messages MAY be changed in non-compatible ways at the time a fork activates, given sufficient time from code-release to activation (as any sane fork would have to have) and there being some in-Template Negotiation Protocol signaling of support for the new fork (e.g. for soft-forks activated using BIP 9).

Field Name	Data Type	Description
template_id	U64	The template_id corresponding to a NewTemplate/RequestTransactionData message.
excess_data	B0_64K	Extra data which the Pool may require to validate the work.
transaction_list	SEQ0_64K[B0_16M]	The transaction data, serialized as a series of B0_16M byte arrays.

RequestTransactionData.Error (Server->Client)

Field Name	Data Type	Description

template_id	U64	The template_id corresponding to a NewTemplate/RequestTransactionData message.
error_code	STR0_255	Reason why no transaction data has been provided

Possible error codes:

- template-id-not-found

SubmitSolution (Client -> Server)

Upon finding a coinbase transaction/nonce pair which double-SHA256 hashes at or below SetNewPrevHash::target, the client MUST immediately send this message, and the server MUST then immediately construct the corresponding full block and attempt to propagate it to the Bitcoin network.

Field Name	Data Type	Description
template_id	U64	The template_id field as it appeared in NewTemplate.
version	U32	The version field in the block header. Bits not defined by BIP320 as additional nonce MUST be the same as they appear in the NewWork message, other bits may be set to any value.
header_timestamp	U32	The nTime field in the block header. This MUST be greater than or equal to the header_timestamp field in the latest SetNewPrevHash message and lower than or equal to that value plus the number of seconds since the receipt of that message.
header_nonce	U32	The nonce field in the header.
coinbase_tx	B0_64K	The full serialized coinbase transaction, meeting all the requirements of the NewWork message, above.

Message Types

Message Type (8-bit)	channel_msg bit	Message Name
0x00	0	SetupConnection
0x01	0	SetupConnection.Success
0x02	0	SetupConnection.Error

0x03	1	ChannelEndpointChanged
Mining protocol		
0x10	0	OpenStandardMiningChannel
0x11	0	OpenStandardMiningChannel.Success
0x12	0	OpenStandardMiningChannel.Error
0x13	0	OpenExtendedMiningChannel
0x14	0	OpenExtendedMiningChannel.Success
0x15	0	OpenExtendedMiningChannel.Error
0x16	1	UpdateChannel
0x17	1	UpdateChannel.Error
0x18	1	CloseChannel
0x19	1	SetExtranoncePrefix
0x1a	1	SubmitSharesStandard
0x1b	1	SubmitSharesExtended
0x1c	1	SubmitShares.Success
0x1d	1	SubmitShares.Error
0x1e	1	NewMiningJob
0x1f	1	NewExtendedMiningJob
0x20	1	SetNewPrevHash
0x21	1	SetTarget
0x22	0	SetCustomMiningJob
0x23	0	SetCustomMiningJob.Success
0x24	0	SetCustomMiningJob.Error
0x25	0	Reconnect
0x26	0	SetGroupChannel
Job Negotiation Protocol		
0x50	0	AllocateMiningJobToken

0x51	0	AllocateMiningJobToken.Success
0x52	0	AllocateMiningJobToken.Error
0x53	0	IdentifyTransactions
0x54	0	IdentifyTransactions.Success
0x55	0	ProvideMissingTransactions
0x56	0	ProvideMissingTransactions.Success
Template Distribution Protocol		
0x70	0	CoinbaseOutputDataSize
0x71	0	NewTemplate
0x72	0	SetNewPrevHash
0x73	0	RequestTransactionData
0x74	0	RequestTransactionData.Success
0x75	0	RequestTransactionData.Error
0x76	0	SubmitSolution

Extensions

There are not yet any defined extensions.

Extension Type (no channel_msg bit)	Extension Name	Description / BIP

Discussion

Speculative Mining Jobs

TBD Describe how exactly sending of new jobs before the next block is found works.

Rolling nTime

nTime field can be rolled once per second with the following notes:

- Mining proxy must not interpret greater than minimum nTime as invalid submission.
- Device MAY roll nTime once per second.
- Pool SHOULD accept nTime which is within the consensus limits.
- Pool MUST accept nTime rolled once per second.

Hardware nTime rolling

The protocol allows nTime rolling in the hardware as long as the hardware can roll the nTime field once per second.

Modern bitcoin ASIC miners do/will support nTime rolling in hardware because it is the most efficient way to expand hashing space for one hashing core/chip. The nTime field is part of the second SHA256 block so it shares midstates with the nonce. Rolling nTime therefore can be implemented as efficiently as rolling nonce, with lowered communication with a mining chip over its communication channels. The protocol needs to allow and support this.

Notes

- Legacy mode: update extranonce1, don't send all the time (send common merkle-root)
- mining on a locally obtained prevhash (server SHOULD queue the work for some time if the miner has faster access to the network).
- Proxying with separated channels helps with merging messages into TCP stream, makes reconnecting more efficient (and done in proxy, not HW), allows to negotiate work for all devices at once.
- Evaluate reaching the design goals.
- Add promise protocol extension support. It is mainly for XMR and ZEC, but can be addressed in generic way already. Promise construction can be coin specific, but the general idea holds for all known use cases.

Usage Scenarios

v2 ST - protocol v2 (this), standard channel

v2 EX - protocol v2, extended channel

v1 - original stratum v1 protocol

End Device (v2 ST)

Typical scenario for end mining devices, header-only mining. The device:

- Sets up the connection without enabling extended channels.
- Opens a Standard channel or more (in the future).
- Receives standard jobs with Merkle root provided by the upstream node.
- Submits standard shares.

Transparent Proxy (v2 any -> v2 any)

Translation from v2 clients to v2 upstream, without aggregating difficulty.

Transparent proxy (connection aggregation):

- Passes all OpenChannel messages from downstream connections to the upstream, with updated request_id for unique identification.
- Associates channel_id given by OpenChannel.Success with the initiating downstream connection. All further messages addressed to the channel_id from the upstream node are passed only to this connection, with channel_id staying stable.

Difficulty Aggregating Proxy (v2 any -> v2 EX)

Proxy:

- Translates all standard ...

V1

(todo difficulty aggregation with info about the devices)

Proxy (v1 -> v2)

Translation from v1 clients to v2 upstream.

The proxy:

- Accept Opens ...

Proxy (v2 -> v1)

...

FAQ

Why is the protocol binary?

The original stratum protocol uses json, which has very bad ratio between the payload size and the actual information transmitted. Designing a binary based protocol yields better data efficiency. Technically, we can use the saved bandwidth for more frequent submits to further reduce the variance in measured hash rate and/or to allow individual machines to submit its work directly instead of using work-splitting mining proxy.

Terminology

- **upstream stratum node**: responsible for providing new mining jobs, information about new prevhash, etc.
- **downstream stratum node**: consumes mining jobs by physically performing proof-of-work computations or by passing jobs onto further downstream devices.
- **channel ID**: identifies an individual mining device or proxy after the channel has been opened. Upstream endpoints perform job submission
- **public key**: ...
- **signature**: signature encoded as...(?)
- **BIP320**: this proposal identifies general purpose bits within version field of bitcoin block header. Mining devices use these bits to extend their search space.
- **Merkle root**: the root hash of a Merkle tree which contains the coinbase transaction and the transaction set consisting of all the other transactions in the block.

Open Questions / Issues

- Write more about channel ID being identifier valid for a particular connection only. It works only in the namespace of it.
- Refresh sequence diagrams.
- Is it useful to have channel-based reconnect? Or is connection-based enough?
- Decide on how to control a single device and allow it to have open multiple channels.
- Describe precisely scenarios with SetNewPrevHash with regards to repeated blockheight
- Decide on how to manage assignment of message ID's and the size of the message ID space. Shall we allow 2 level multiplexing? E.g. dedicate an ID to a class of vendor messages, while allowing the vendor to assign custom messages ID's within the class?

- More information about telemetry data

Hashing Power Information		
Field Name	**Data Type**	**Description**
aggregated_device_count	U32	Number of aggregated devices on the channel. An end mining device must send 1. A proxy can send 0 when there are no connections to it yet (in aggregating mode)