```
BIP: 152
Layer: Peer Services
Title: Compact Block Relay
Author: Matt Corallo <bip152@bluematt.me>
Comments-Summary: Unanimously Recommended for implementation
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0152
Status: Final
Type: Standards Track
Created: 2016-04-27
License: PD
```

Abstract

Compact blocks on the wire as a way to save bandwidth for nodes on the P2P network.

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.

Motivation

Historically, the Bitcoin P2P protocol has not been very bandwidth efficient for block relay. Every transaction in a block is included when relayed, even though a large number of the transactions in a given block are already available to nodes before the block is relayed. This causes moderate inbound bandwidth spikes for nodes when receiving blocks, but can cause very significant outbound bandwidth spikes for some nodes which receive a block before their peers. When such spikes occur, buffer bloat can make consumer-grade internet connections temporarily unusable, and can delay the relay of blocks to remote peers who may choose to wait instead of redundantly requesting the same block from other, less congested, peers.

Thus, decreasing the bandwidth used during block relay is very useful for many individuals running nodes.

While the goal of this work is explicitly not to reduce block transfer latency, it does, as a side effect reduce block transfer latencies in some rather significant ways. Additionally, this work forms a foundation for future work explicitly targeting low-latency block transfer.

Specification for version 1

Intended Protocol Flow

The protocol is intended to be used in two ways, depending on the peers and bandwidth available, as discussed later. The "high-bandwidth" mode, which nodes may only enable for a few of their peers, is enabled by setting the first boolean to

1 in a `sendcmpct` message. In this mode, peers send new block announcements with the short transaction IDs already (via a `cmpctblock` message), possibly even before fully validating the block (as indicated by the grey box in the image above). In some cases no further round-trip is needed, and the receiver can reconstruct the block and process it as usual immediately. When some transactions were not available from local sources (ie mempool), a `getblocktxn/blocktxn` roundtrip is necessary, bringing the best-case latency to the same 1.5*RTT minimum time that nodes take today, though with significantly less bandwidth usage.

The "low-bandwidth" mode is enabled by setting the first boolean to 0 in a `sendcmpct` message. In this mode, peers send new block announcements with the usual inv/headers announcements (as per BIP130, and after fully validating the block). The receiving peer may then request the block using a MSG_CMPCT_BLOCK `getdata` request, which will receive a response of the header and short transaction IDs. In some cases no further round-trip is needed, and the receiver can reconstruct the block and process it as usual, taking the same 1.5*RTT minimum time that nodes take today, though with significantly less bandwidth usage. When some transactions were not available from local sources (ie mempool), a `getblocktxn/blocktxn` roundtrip is necessary, bringing the latency to at least 2.5*RTT in this case, again with significantly less bandwidth usage than today. Because TCP often exhibits worse transfer latency for larger data sizes (as a multiple of RTT), total latency is expected to be reduced even when the full 2.5*RTT transfer mechanism is used.

New data structures

Several new data structures are added to the P2P network to relay compact blocks: PrefilledTransaction, HeaderAndShortIDs, BlockTransactionsRequest, and BlockTransactions.

For the purposes of this section, CompactSize refers to the variable-length integer encoding used across the existing P2P protocol to encode array lengths, among other things, in 1, 3, 5 or 9 bytes. Only CompactSize encodings which are minimally-encoded (ie the shortest length possible) are used by this spec. Any other CompactSize encodings are left with undefined behavior.

Several uses of CompactSize below are "differentially encoded". For these, instead of using raw indexes, the number encoded is the difference between the current index and the previous index, minus one. For example, a first index of 0 implies a real index of 0, a second index of 0 thereafter refers to a real index of 1, etc.

PrefilledTransaction A PrefilledTransaction structure is used in HeaderAndShortIDs to provide a list of a few transactions explicitly.

Field Name	Type	Size	Encoding
index	CompactSize	1, 3 bytes	Compact Size, differentially encoded since the last PrefilledTransac
tx	Transaction	variable	As encoded in "tx" messages sent in response to getdata MSG_TX

HeaderAndShortIDs A HeaderAndShortIDs structure is used to relay a block header, the short transactions IDs used for matching already-available transactions, and a select few transactions which we expect a peer may be missing.

Field Name	Type	Size	Encoding
header	Block header	80 bytes	First 80 bytes of the
nonce	uint64_t	8 bytes	Little Endian
shortids_length	CompactSize	1 or 3 bytes	As used to encode ar
shortids	List of 6-byte integers	6*shortids_length bytes	Little Endian
prefilledtxn_length	CompactSize	1 or 3 bytes	As used to encode ar
prefilledtxn	List of PrefilledTransactions	variable size*prefilledtxn_length	As defined by Prefille

BlockTransactionsRequest A BlockTransactionsRequest structure is used to list transaction indexes in a block being requested.

Field Name	Type	Size	Encoding
blockhash	Binary blob	32 bytes	The output from a double-SHA256 o
indexes_length	CompactSize	1 or 3 bytes	As used to encode array lengths elsev
indexes	List of CompactSizes	1 or 3 bytes*indexes_length	Differentially encoded

BlockTransactions A BlockTransactions structure is used to provide some of the transactions in a block, as requested.

Field Name	Type	Size	Encoding
blockhash	Binary blob	32 bytes	The output from a double-SHA256 of the block h
transactions_length	CompactSize	1 or 3 bytes	As used to encode array lengths elsewhere
transactions	List of Transactions	variable	As encoded in "tx" messages in response to getdat

Short transaction IDs Short transaction IDs are used to represent a transaction without sending a full 256-bit hash. They are calculated by:

1. single-SHA256 hashing the block header with the nonce appended (in little-endian)
2. Running SipHash-2-4 with the input being the transaction ID and the keys (k0/k1) set to the first two little-endian 64-bit integers from the above hash, respectively.
3. Dropping the 2 most significant bytes from the SipHash output to make it 6 bytes.

New messages

A new inv type (MSG_CMPCT_BLOCK == 4) and several new protocol messages are added: sendcmpct, cmpctblock, getblocktxn, and blocktxn.

sendcmpct

1. The sendcmpct message is defined as a message containing a 1-byte integer followed by a 8-byte integer where pchCommand == "sendcmpct".
2. The first integer SHALL be interpreted as a boolean (and MUST have a value of either 1 or 0)
3. The second integer SHALL be interpreted as a little-endian version number. Nodes sending a sendcmpct message MUST currently set this value to 1.
4. Upon receipt of a "sendcmpct" message with the first and second integers set to 1, the node SHOULD announce new blocks by sending a cmpctblock message.
5. Upon receipt of a "sendcmpct" message with the first integer set to 0, the node SHOULD NOT announce new blocks by sending a cmpctblock message, but SHOULD announce new blocks by sending invs or headers, as defined by BIP130.
6. Upon receipt of a "sendcmpct" message with the second integer set to something other than 1, nodes MUST treat the peer as if they had not received the message (as it indicates the peer will provide an unexpected encoding in cmpctblock, and/or other, messages). This allows future versions to send duplicate sendcmpct messages with different versions as a part of a version handshake for future versions. See Protocol Versioning section, below, for more info on the specifics of the version-negotiation mechanics.
7. Nodes SHOULD check for a protocol version of >= 70014 before sending sendcmpct messages.
8. Nodes MUST NOT send a request for a MSG_CMPCT_BLOCK object to a peer before having received a sendcmpct message from that peer.
9. Nodes MUST NOT request a MSG_CMPCT_BLOCK object before having sent all sendcmpct messages to that peer which they intend to send, as the peer cannot know what version protocol to use in the response.

MSG_CMPCT_BLOCK

1. getdata messages may now contain requests for MSG_CMPCT_BLOCK objects.
2. Upon receipt of a getdata containing a request for a MSG_CMPCT_BLOCK object with the hash of a block which was recently announced and is close to the tip of the best chain of the receiver and after having sent the requesting peer a sendcmpct message, nodes MUST respond with a cmpctblock message containing appropriate data representing the block being requested.
3. Upon receipt of a getdata containing a request for a MSG_CMPCT_BLOCK object for which a cmpctblock message is not sent in response, a block message containing the requested block in non-compact form MUST be sent.
4. MSG_CMPCT_BLOCK inv objects MUST NOT appear anywhere except

for in getdata messages.

cmpctblock

1. The cmpctblock message is defined as a message containing a serialized HeaderAndShortIDs message and pchCommand == "cmpctblock".
2. Upon receipt of a cmpctblock message after sending a sendcmpct message, nodes SHOULD calculate the short transaction ID for each unconfirmed transaction they have available (ie in their mempool) and compare each to each short transaction ID in the cmpctblock message.
3. After finding already-available transactions, nodes which do not have all transactions available to reconstruct the full block SHOULD request the missing transactions using a getblocktxn message.
4. A node MUST NOT send a cmpctblock message unless they are able to respond to a getblocktxn message which requests every transaction in the block.
5. A node MUST NOT send a cmpctblock message without having validated that the header properly commits to each transaction in the block, and properly builds on top of the existing, fully-validated chain with a valid proof-of-work either as a part of the current most-work valid chain, or building directly on top of it. A node MAY send a cmpctblock before validating that each transaction in the block validly spends existing UTXO set entries.

getblocktxn

1. The getblocktxn message is defined as a message containing a serialized BlockTransactionsRequest message and pchCommand == "getblocktxn".
2. Upon receipt of a properly-formatted getblocktxn message, nodes which recently provided the sender of such a message a cmpctblock for the block hash identified in this message MUST respond with either an appropriate blocktxn message, or a full block message. A blocktxn response MUST contain exactly and only each transaction which is present in the appropriate block at the index specified in the getblocktxn indexes list, in the order requested.

blocktxn

1. The blocktxn message is defined as a message containing a serialized BlockTransactions message and pchCommand == "blocktxn".
2. Upon receipt of a properly-formatted requested blocktxn message, nodes SHOULD attempt to reconstruct the full block by:
   (a) Taking the prefilledtxn transactions from the original cmpctblock and placing them in the marked positions.
   (b) For each short transaction ID from the original cmpctblock, in order, find the corresponding transaction either from the blocktxn message

or from other sources and place it in the first available position in the block.

3. Once the block has been reconstructed, it shall be processed as normal, keeping in mind that short transaction IDs are expected to occasionally collide, and that nodes MUST NOT be penalized for such collisions, wherever they appear.

Protocol Versioning

1. The protocol version negotiation allows two nodes to agree on the versions of compact blocks which they will exchange. As it is only in a single field, it does not allow a node to support a specific version in only one direction (sending or receiving).
2. Upon connection establishment, a node SHOULD send a burst of sendcmpct messages containing every version of compact block encodings for which they are willing to support sending cmpctblock and blocktxn messages, and receiving getblocktxn messages. These messages SHOULD be ordered in the order of the priority which the node wishes to receive cmpctblock/blocktxn messages, with the highest-priority version sendcmpct message sent first.
3. The encoding version used to send a cmpctblock or blocktxn message or to receive a getblocktxn message MUST be the second integer (version number) in the first sendcmpct message received for which a sendcmpct message with the same version number was sent.
4. Nodes MUST NOT send a sendcmpct message which contains a version number other than the version number which has been negotiated for receiving cmpctblock/blocktxn messages after sending a request for a MSG_CMPCT_BLOCK object, sending a cmpctblock, getblocktxn, blocktxn, or pong message.
5. As a node must send all sendcmpct messages which contain a novel version announcement before any other compact block-related messages, it is possible to determine which version of compact blocks will be used for each object received. It is, however, not possible to know which version will be used to encode the response to a request for a compact block object before any MSG_CMPCT_BLOCK-containing getdata, cmpctblock, getblocktxn, blocktxn, or ping/pong messages have been exchanged.
6. Thus, if a node wishes to determine exactly which version of compact blocks will be used before requesting a compact block object, it must send all of its sendcmpct version announcements, followed by a ping, and wait for the pong response to ensure it has received all sendcmpctblock version announcement messages from the remote peer. Nodes can, obviously, however, determine that the version used will be at least a certain version (in their priority order) after having received a sendcmpct message from the remote peer containing that version as the second integer.

Sample Version Implementation

1. By way of example, an implementation of the above protocol might look like the following.
2. Upon exchanging version/verack messages, a node immediately sends its list of sendcmpct announcements to the other side, with the version which it wants to receive sent first.
3. Upon receiving the first sendcmpct announcement with a protocol version which is understood from the remote peer, a node will "lock in" the compact block encoding version which will be used to encode compact blocks to that peer.
4. The node then sets the current receive-protocol-version in use on the connection to that version, and uses it to decode new compact block messages.
5. Upon receiving subsequent sendcmpct announcements with a protocol version which is understood from the remote peer (ie a version which has been announced using a sendcmpct in the other direction), a node will check if that protocol version is higher-receive-priority than the current receive-protocol-version in use on the connection, and switch to that version for decoding new compact block messages received.
6. A node might wish to keep a flag for each peer which indicates compact block version negotiation is complete, which can be set upon receiving any compact block-related, or pong message.
7. The above implementation requires only a compile-time list of supported versions in some static priority order, two version fields per peer, and an optional negotiation-complete boolean per-peer.

Specification for version 2

Compact blocks version 2 is almost identical to version 1, but supports segregated witness transactions (BIP 141 and BIP 144). The changes are:

1. The second integer (version number) inside sendcmpct is 2 instead of 1 (see Protocol Versioning section, above).
2. Transactions inside cmpctblock messages (both those used as direct announcement and those in response to getdata) and in blocktxn should include witness data, using the same format as responses to getdata MSG_WITNESS_TX, specified in BIP144.
3. Short transaction IDs sent to us in cmpctblock messages, and sent by us in getblocktxn messages, are computed using the same process as in version 1, but using the wtxid as defined in BIP 141 instead of the txid. Note that, though a node normally SHOULD, if a node does not include (ie must then include the short ID for) the coinbase transaction, it must be computed by encoding the transaction in witness format as defined by BIP 141.
4. Upon receipt of a getdata containing a request for a MSG_CMPCT_BLOCK object for which a cmpctblock message is not sent in response, the block message containing the requested block in non-compact form MUST be

encoded with witnesses (as is sent in reply to a MSG_WITNESS_BLOCK getdata) if the protocol version used to encode the cmpctblock message would have been 2, and encoded without witnesses (as is sent in response to a MSG_BLOCK getdata) if the protocol version used to encode the cmpctblock message would have been 1.

Implementation Notes

1. For nodes which have sufficient inbound bandwidth, sending a sendcmpct message with the first integer set to 1 to up to 3 peers is RECOMMENDED. If possible, it is RECOMMENDED that those peers be selected based on their past performance in providing blocks quickly (eg the three peers which provided the highest number of the recent N blocks the quickest), allowing nodes to receive blocks which come from those peers in only 0.5*RTT.

1. Nodes MUST NOT send such sendcmpct messages to more than three peers, as it encourages wasting outbound bandwidth across the network.

1. All nodes SHOULD send a sendcmpct message to all appropriate peers. This will reduce their outbound bandwidth usage by allowing their peers to request compact blocks instead of full blocks.

1. Nodes with limited inbound bandwidth SHOULD request blocks using MSG_CMPCT_BLOCK/getblocktxn requests, when possible. While this increases worst-case message round-trips, it is expected to reduce overall transfer latency as TCP is more likely to exhibit poor throughput on low-bandwidth nodes.

1. Nodes sending cmpctblock messages SHOULD limit prefilledtxn to 10KB of transactions. When in doubt, nodes SHOULD only include the coinbase transaction in prefilledtxn.

1. Nodes MAY pick one nonce per block they wish to send, and only build a cmpctblock message once for all peers which they wish to send a given block to. Nodes SHOULD NOT use the same nonce across multiple different blocks.

1. Nodes MAY impose additional requirements on when they announce new blocks by sending cmpctblock messages. For example, nodes with limited outbound bandwidth MAY choose to announce new blocks using inv/header messages (as per BIP130) to conserve outbound bandwidth.

1. Note that the MSG_CMPCT_BLOCK section does not require that nodes respond to MSG_CMPCT_BLOCK getdata requests for blocks which they did not recently announce. This allows nodes to calculate cmpctblock messages at announce-time instead of at request-time. Blocks which are requested with a MSG_CMPCT_BLOCK getdata, but which are not responded to with a cmpctblock message MUST be responded to with a block message, allowing nodes to request all blocks using

MSG_CMPCT_BLOCK getdatas and rely on their peer to pick an appropriate response.

1. While the current version sends transactions with the same encodings as are used in tx messages and elsewhere in the protocol, the version field in sendcmpct is intended to allow this to change in the future. For this reason, it is recommended that the code used to decode PrefilledTransaction and BlockTransactions messages be prepared to take a different transaction encoding, if and when the version field in sendcmpct changes in a future BIP.

1. Any undefined behavior in this spec may cause failure to transfer block to, peer disconnection by, or self-destruction by the receiving node. A node receiving non-minimally-encoded CompactSize encodings should make a best-effort to eat the sender's cat.

Pre-Validation Relay and Consistency Considerations

1. As high-bandwidth mode permits relaying of CMPCTBLOCK messages prior to full validation (requiring only that the block header is valid before relay), nodes SHOULD NOT ban a peer for announcing a new block with a CMPCTBLOCK message that is invalid, but has a valid header. For avoidance of doubt, nodes SHOULD bump their peer-to-peer protocol version to 70015 or higher to signal that they will not ban or punish a peer for announcing compact blocks prior to full validation, and nodes SHOULD NOT announce a CMPCTBLOCK to a peer with a version number below 70015 before fully validating the block.

1. SPV nodes which implement this spec must consider the implications of accepting blocks which were not validated by the node which provided them. Especially SPV nodes which allow users to select a "trusted full node" to sync from may wish to avoid implementing this spec in high-bandwidth mode.

1. Note that this spec does not change the requirement that nodes only relay information about blocks which they have fully validated in response to GETDATA/GETHEADERS/GETBLOCKS/etc requests. Nodes which announce using CMPCTBLOCK message and then receive a request for associated block data SHOULD ensure that messages do not go unresponded to, and that the appropriate data is provided after the block has been validated, subject to standard message-response ordering requirements. Note that no requirement is added that the node respond to the request with the new block included in eg GETHEADERS or GETBLOCKS messages, but the node SHOULD re-announce the block using the associated announcement methods after validation has completed if it is not included in the original response. On the other hand, nodes SHOULD delay responding to GETDATA requests for the block until validation has completed, stalling all message processing for the associated peer. REJECT messages are not

considered "responses" for the purpose of this section.

1. As a result of the above requirements, implementors may wish to consider the potential for the introduction of delays in responses while remote peers validate blocks, avoiding delay-causing requests where possible.

Justification

Protocol design There have been many proposals to save wire bytes when relaying blocks. Many of them have a two-fold goal of reducing block relay time and thus rely on the use of significant processing power in order to avoid introducing additional worst-case RTTs. Because this work is not focused primarily on reducing block relay time, its design is much simpler (ie does not rely on set reconciliation protocols). Still, in testing at the time of writing, nodes are able to relay blocks without the extra getblocktxn/blocktxn RTT around 90% of the time. With a smart compact-block-announcement policy, it is thus expected that this work might allow blocks to be relayed between nodes in 0.5*RTT instead of 1.5*RTT at least 75% of the time.

Short transaction ID calculation There are several design goals for the Short ID calculation:

- **Performance** The sender needs to compute short IDs for all block transactions, and the receiver for all mempool transactions they are being compared to. As we're easily talking about several thousand transactions, sub-microsecond processing per-transactions is needed.
- **Space** cmpctblock messages are never optional in this protocol, and contain a short ID for each non-prefilled transaction in the block. Thus, the size of short IDs is directly proportional to the maximum bandwidth savings possible.
- **Collision resistance** It should be hard for network participants to create transactions that cause collisions. If an attacker were able to cause such collisions, filling mempools (and, thus, blocks) with them would cause poor network propagation of new (or non-attacker, in the case of a miner) blocks.

SipHash is a secure, fast, and simple 64-bit MAC designed for network traffic authentication and collision-resistant hash tables. We truncate the output from SipHash-2-4 to 48 bits (see next section) in order to minimize space. The resulting 48-bit hash is certainly not large enough to avoid intentionally created individual collisons, but by using the block hash as a key to SipHash, an attacker cannot predict what keys will be used once their transactions are actually included in a relayed block. We mix in a per-connection 64-bit nonce to obtain independent short IDs on every connection, so that even block creators cannot control where collisions occur, and random collisions only ever affect a small number of connections at any given time. The mixing is done using SHA256(block_header || nonce), which is slow compared to SipHash, but only

done once per block. It also adds the ability for nodes to choose the nonce in a better than random way to minimize collisions, though that is not necessary for correct behaviour. Conversely, nodes can also abuse this ability to increase their ability to introduce collisions in the blocks they relay themselves. However, they can already cause more problems by simply refusing to relay blocks. That is inevitable, and this design only seeks to prevent network-wide misbehavior.

Random collision probabilty Thanks to the block-header-based SipHash keys, we can assume that the only collisions on links between honest nodes are random ones.

For each of the t block transactions, the receiver will compare its received short ID with that of a set of m mempool transactions. We assume that each of those t has a chance r to be included in that set of m. If we use B bits short IDs, for each comparison between a received short ID and a mempool transaction, there is a chance of $P = 1 - 1 / 2^B$ that a mismatch is detected as such.

When comparing a given block transaction to the whole set of mempool transactions, there are 5 cases to distinguish:

1. The receiver has exactly one match, which is the correct one. This has chance $r * P^(m - 1)$.
2. The receiver has no matches. This has chance $(1 - r) * P^m$.
3. The receiver has at least two matches, one of which is correct. This has chance $r * (1 - P^(m - 1))$.
4. The receiver has at least two matches, both of which are incorrect. This has chance $(1 - r) * (1 - P^m - m * (1 - P) * P^(m - 1))$.
5. The receiver has exactly one match, but an incorrect one. This has chance $(1 - r) * m * (1 - P) * P^(m - 1)$.

(note that these 5 numbers always add up to 100%)

In case 1, we're good. In cases 2, 3, or 4, we request the full transaction because we know we're uncertain. Only in case 5, we fail to reconstruct. The chance that case 5 does not occur in any of the t transactions in a block is $(1 - (1 - r) * m * (1 - P) * P^(m - 1))^t$. This expression is well approximated by $1 - (1 - r) * m * (1 - P) * t = 1 - (1 - r) * m * t / 2^B$. Thus, if we want only one in F block transmissions between honest nodes to fail under the conservative $r = 0$ assumption, we need $log2(F * m * t)$ bits hash functions.

This means that $B = 48$ bits short IDs suffice for blocks with up to $t = 10000$ transactions, mempools up to $m = 100000$ transactions, with failure to reconstruct at most one in $F = 281474$ blocks. Since failure to reconstruct just means we fall back to normal inv/header based relay, it isn't necessary to avoid such failure completely. It just needs to be sufficiently rare they have a lower impact than random transmission failures (for example, network disconnection, node overloaded, ...).

Separate version for segregated witness The changes to transaction and block relay in BIP 144 introduce separate MSG_FILTERED_ versions of messages in getdata, allowing a receiver to choose individually where witness data is wanted.

This method is not useful for compact blocks because ‘cmpctblock‘ blocks can be sent unsolicitedly in high-bandwidth mode, so we need to negotiate at least whether those should include witness data up front. There is little use for a validating node that only sometimes processes witness data, so we may as well use that negotiation for everything and turn it into a separate protocol version. We also need a means to distinguish different versions of the same transaction with different witnesses for correct reconstruction, so this also forces us to use wtxids instead of txids for short IDs everywhere in that case.

Backward compatibility

Older clients remain fully compatible and interoperable after this change.

Implementation

https://github.com/bitcoin/bitcoin/pull/8068 for version 1. https://github.com/bitcoin/bitcoin/pull/8393 for version 2.

Acknowledgements

Thanks to Gregory Maxwell for the initial suggestion as well as a lot of back-and-forth design and significant testing. Thanks to Nicolas Dorier for the protocol flow diagram.

Copyright