```
BIP: 37
Layer: Peer Services
Title: Connection Bloom filtering
Author: Mike Hearn <hearn@google.com>
        Matt Corallo <bip37@bluematt.me>
Comments-Summary: No comments yet.
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0037
Status: Final
Type: Standards Track
Created: 2012-10-24
License: PD
```

Abstract

This BIP adds new support to the peer-to-peer protocol that allows peers to reduce the amount of transaction data they are sent. Peers have the option of setting *filters* on each connection they make after the version handshake has completed. A filter is defined as a Bloom filter on data derived from transactions. A Bloom filter is a probabilistic data structure which allows for testing set membership - they can have false positives but not false negatives.

This document will not go into the details of how Bloom filters work and the reader is referred to Wikipedia for an introduction to the topic.

Motivation

As Bitcoin grows in usage the amount of bandwidth needed to download blocks and transaction broadcasts increases. Clients implementing *simplified payment verification* do not attempt to fully verify the block chain, instead just checking that block headers connect together correctly and trusting that the transactions in a chain of high difficulty are in fact valid. See the Bitcoin paper for more detail on this mode.

Today, SPV clients have to download the entire contents of blocks and all broadcast transactions, only to throw away the vast majority of the transactions that are not relevant to their wallets. This slows down their synchronization process, wastes users bandwidth (which on phones is often metered) and increases memory usage. All three problems are triggering real user complaints for the Android "Bitcoin Wallet" app which implements SPV mode. In order to make chain synchronization fast, cheap and able to run on older phones with limited memory we want to have remote peers throw away irrelevant transactions before sending them across the network.

Design rationale

The most obvious way to implement the stated goal would be for clients to upload lists of their keys to the remote node. We take a more complex approach for the following reasons:

- Privacy: Because Bloom filters are probabilistic, with the false positive rate chosen by the client, nodes can trade off precision vs bandwidth usage. A node with access to lots of bandwidth may choose to have a high FP rate, meaning the remote peer cannot accurately know which transactions belong to the client and which don't. A node with very little bandwidth may choose to use a very accurate filter meaning that they only get sent transactions actually relevant to their wallet, but remote peers may be able to correlate transactions with IP addresses (and each other).
- Bloom filters are compact and testing membership in them is fast. This results in satisfying performance characteristics with minimal risk of opening up potential for DoS attacks.

Specification

New messages

We start by adding three new messages to the protocol:

- `filterload`, which sets the current Bloom filter on the connection
- `filteradd`, which adds the given data element to the connections current filter without requiring a completely new one to be set
- `filterclear`, which deletes the current filter and goes back to regular pre-BIP37 usage.

Note that there is no filterremove command because by their nature, Bloom filters are append-only data structures. Once an element is added it cannot be removed again without rebuilding the entire structure from scratch.

The `filterload` command is defined as follows:

Field Size	Description	Data type	Comments
?	filter	uint8_t[]	The filter itself is simply a bit field of arbitrary byte-aligned size. The
4	nHashFuncs	uint32_t	The number of hash functions to use in this filter. The maximum valu
4	nTweak	uint32_t	A random value to add to the seed value in the hash function used by
1	nFlags	uint8_t	A set of flags that control how matched items are added to the filter.

See below for a description of the Bloom filter algorithm and how to select nHashFuncs and filter size for a desired false positive rate.

Upon receiving a `filterload` command, the remote peer will immediately restrict the broadcast transactions it announces (in inv packets) to transactions matching the filter, where the matching algorithm is specified below. The flags control the update behaviour of the matching algorithm.

The `filteradd` command is defined as follows:

Field Size	Description	Data type	Comments
?	data	uint8_t[]	The data element to add to the current filter.

The data field must be smaller than or equal to 520 bytes in size (the maximum size of any potentially matched object).

The given data element will be added to the Bloom filter. A filter must have been previously provided using `filterload`. This command is useful if a new key or script is added to a clients wallet whilst it has connections to the network open, it avoids the need to re-calculate and send an entirely new filter to every peer (though doing so is usually advisable to maintain anonymity).

The `filterclear` command has no arguments at all.

After a filter has been set, nodes don't merely stop announcing non-matching transactions, they can also serve filtered blocks. A filtered block is defined by the `merkleblock` message and is defined like this:

Field Size	Description	Data type	Comments
4	version	uint32_t	Block version information, based upon the software version crea
32	prev_block	char[32]	The hash value of the previous block this particular block refer
32	merkle_root	char[32]	The reference to a Merkle tree collection which is a hash of all
4	timestamp	uint32_t	A timestamp recording when this block was created (Limited t
4	bits	uint32_t	The calculated difficulty target being used for this block
4	nonce	uint32_t	The nonce used to generate this block… to allow variations of
4	total_transactions	uint32_t	Number of transactions in the block (including unmatched ones
?	hashes	uint256[]	hashes in depth-first order (including standard varint size prefi
?	flags	byte[]	flag bits, packed per 8 in a byte, least significant bit first (inclu

See below for the format of the partial merkle tree hashes and flags.

Thus, a `merkleblock` message is a block header, plus a part of a merkle tree which can be used to extract identifying information for transactions that matched the filter and prove that the matching transaction data really did appear in the solved block. Clients can use this data to be sure that the remote node is not feeding them fake transactions that never appeared in a real block, although lying through omission is still possible.

Extensions to existing messages

The `version` command is extended with a new field:

Field Size	Description	Data type	Comments
1 byte	fRelay	bool	If false then broadcast transactions will not be announced until a filte

SPV clients that wish to use Bloom filtering would normally set fRelay to false in the version message, then set a filter based on their wallet (or a subset of it, if they are overlapping different peers). Being able to opt-out of inv messages until the filter is set prevents a client being flooded with traffic in the brief window of time between finishing version handshaking and setting the filter.

The `getdata` command is extended to allow a new type in the `inv` submessage. The type field can now be `MSG_FILTERED_BLOCK (== 3)` rather than `MSG_BLOCK`. If no filter has been set on the connection, a request for filtered blocks is ignored. If a filter has been set, a `merkleblock` message is returned for the requested block hash. In addition, because a `merkleblock` message contains only a list of transaction hashes, transactions matching the filter should also be sent in separate tx messages after the merkleblock is sent. This avoids a slow roundtrip that would otherwise be required (receive hashes, didn't see some of these transactions yet, ask for them). Note that because there is currently no way to request transactions which are already in a block from a node (aside from requesting the full block), the set of matching transactions that the requesting node hasn't either received or announced with an inv must be sent and any additional transactions which match the filter may also be sent. This allows for clients (such as the reference client) to limit the number of invs it must remember a given node to have announced while still providing nodes with, at a minimum, all the transactions it needs.

Filter matching algorithm

The filter can be tested against arbitrary pieces of data, to see if that data was inserted by the client. Therefore the question arises of what pieces of data should be inserted/tested.

To determine if a transaction matches the filter, the following algorithm is used. Once a match is found the algorithm aborts.

1. Test the hash of the transaction itself.
2. For each output, test each data element of the output script. This means each hash and key in the output script is tested independently. **Important:** if an output matches whilst testing a transaction, the node might need to update the filter by inserting the serialized COutPoint structure. See below for more details.
3. For each input, test the serialized COutPoint structure.
4. For each input, test each data element of the input script (note: input scripts only ever contain data elements).
5. Otherwise there is no match.

In this way addresses, keys and script hashes (for P2SH outputs) can all be added to the filter. You can also match against classes of transactions that are marked with well known data elements in either inputs or outputs, for example, to implement various forms of Smart property.

The test for outpoints is there to ensure you can find transactions spending outputs in your wallet, even though you don't know anything about their form. As you can see, once set on a connection the filter is **not static** and can change throughout the connections lifetime. This is done to avoid the following race condition:

1. A client sets a filter matching a key in their wallet. They then start downloading the block chain. The part of the chain that the client is missing is requested using getblocks.
2. The first block is read from disk by the serving peer. It contains TX 1 which sends money to the clients key. It matches the filter and is thus sent to the client.
3. The second block is read from disk by the serving peer. It contains TX 2 which spends TX 1. However TX 2 does not contain any of the clients keys and is thus not sent. The client does not know the money they received was already spent.

By updating the bloom filter atomically in step 2 with the discovered outpoint, the filter will match against TX 2 in step 3 and the client will learn about all relevant transactions, despite that there is no pause between the node processing the first and second blocks.

The nFlags field of the filter controls the nodes precise update behaviour and is a bit field.

- `BLOOM_UPDATE_NONE (0)` means the filter is not adjusted when a match is found.
- `BLOOM_UPDATE_ALL (1)` means if the filter matches any data element in a scriptPubKey the outpoint is serialized and inserted into the filter.
- `BLOOM_UPDATE_P2PUBKEY_ONLY (2)` means the outpoint is inserted into the filter only if a data element in the scriptPubKey is matched, and that script is of the standard "pay to pubkey" or "pay to multisig" forms.

These distinctions are useful to avoid too-rapid degradation of the filter due to an increasing false positive rate. We can observe that a wallet which expects to receive only payments of the standard pay-to-address form doesn't need automatic filter updates because any transaction that spends one of its own outputs has a predictable data element in the input (the pubkey that hashes to the address). If a wallet might receive pay-to-address outputs and also pay-to-pubkey or pay-to-multisig outputs then BLOOM_UPDATE_P2PUBKEY_ONLY is appropriate, as it avoids unnecessary expansions of the filter for the most common types of output but still ensures correct behaviour with payments that explicitly specify keys.

Obviously, nFlags == 1 or nFlags == 2 mean that the filter will get dirtier as more of the chain is scanned. Clients should monitor the observed false positive rate and periodically refresh the filter with a clean one.

Partial Merkle branch format

A *Merkle tree* is a way of arranging a set of items as leaf nodes of tree in which the interior nodes are hashes of the concatenations of their child hashes. The root node is called the *Merkle root.* Every Bitcoin block contains a Merkle root of the tree formed from the blocks transactions. By providing some elements of the trees interior nodes (called a *Merkle branch*) a proof is formed that the given transaction was indeed in the block when it was being mined, but the size of the proof is much smaller than the size of the original block.

Constructing a partial merkle tree object

- Traverse the merkle tree from the root down, and for each encountered node:
  - Check whether this node corresponds to a leaf node (transaction) that is to be included OR any parent thereof:
    - If so, append a '1' bit to the flag bits
    - Otherwise, append a '0' bit.
  - Check whether this node is a internal node (non-leaf) AND is the parent of an included leaf node:
    - If so:
      - Descend into its left child node, and process the subtree beneath it entirely (depth-first).
      - If this node has a right child node too, descend into it as well.
    - Otherwise: append this node's hash to the hash list.

Parsing a partial merkle tree object As the partial block message contains the number of transactions in the entire block, the shape of the merkle tree is known before hand. Again, traverse this tree, computing traversed node's hashes along the way:

- Read a bit from the flag bit list:
  - If it is '0':
    - Read a hash from the hashes list, and return it as this node's hash.
  - If it is '1' and this is a leaf node:
    - Read a hash from the hashes list, store it as a matched txid, and return it as this node's hash.
  - If it is '1' and this is an internal node:
    - Descend into its left child tree, and store its computed hash as L.
    - If this node has a right child as well:
      - Descend into its right child, and store its computed hash as R.
      - If L == R, the partial merkle tree object is invalid.
      - Return Hash(L || R).
    - If this node has no right child, return Hash(L || L).

The partial merkle tree object is only valid if:

- All hashes in the hash list were consumed and no more.
- All bits in the flag bits list were consumed (except padding to make it into a full byte), and no more.
- The hash computed for the root node matches the block header's merkle root.
- The block header is valid, and matches its claimed proof of work.
- In two-child nodes, the hash of the left and right branches was never equal.

Bloom filter format

A Bloom filter is a bit-field in which bits are set based on feeding the data element to a set of different hash functions. The number of hash functions used is a parameter of the filter. In Bitcoin we use version 3 of the 32-bit Murmur hash function. To get N "different" hash functions we simply initialize the Murmur algorithm with the following formula:

`nHashNum * 0xFBA4C795 + nTweak`

i.e. if the filter is initialized with 4 hash functions and a tweak of 0x00000005, when the second function (index 1) is needed h1 would be equal to 4221880218.

When loading a filter with the `filterload` command, there are two parameters that can be chosen. One is the size of the filter in bytes. The other is the number of hash functions to use. To select the parameters you can use the following formulas:

Let N be the number of elements you wish to insert into the set and P be the probability of a false positive, where 1.0 is "match everything" and zero is unachievable.

The size S of the filter in bytes is given by `(-1 / pow(log(2), 2) * N * log(P)) / 8`. Of course you must ensure it does not go over the maximum size (36,000: selected as it represents a filter of 20,000 items with false positive rate of < 0.1% or 10,000 items and a false positive rate of < 0.0001%).

The number of hash functions required is given by `S * 8 / N * log(2)`.

Copyright