The Bitcoin Network

Merkle blocks and Bloom filters

Bitcoin network

The references you need to know everything here:

https://en.bitcoin.it/wiki/Protocol_documentation

https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki
(also contains the solution to Homework 1 ☺)

Bitcoin network

Up to now:

- We know how to connect to a full node
- We know how to receive block headers
- We know how to chekc the proof-of-work

This class:

- We will simulate what an SPV node does in terms of network communication

Bitcoin network

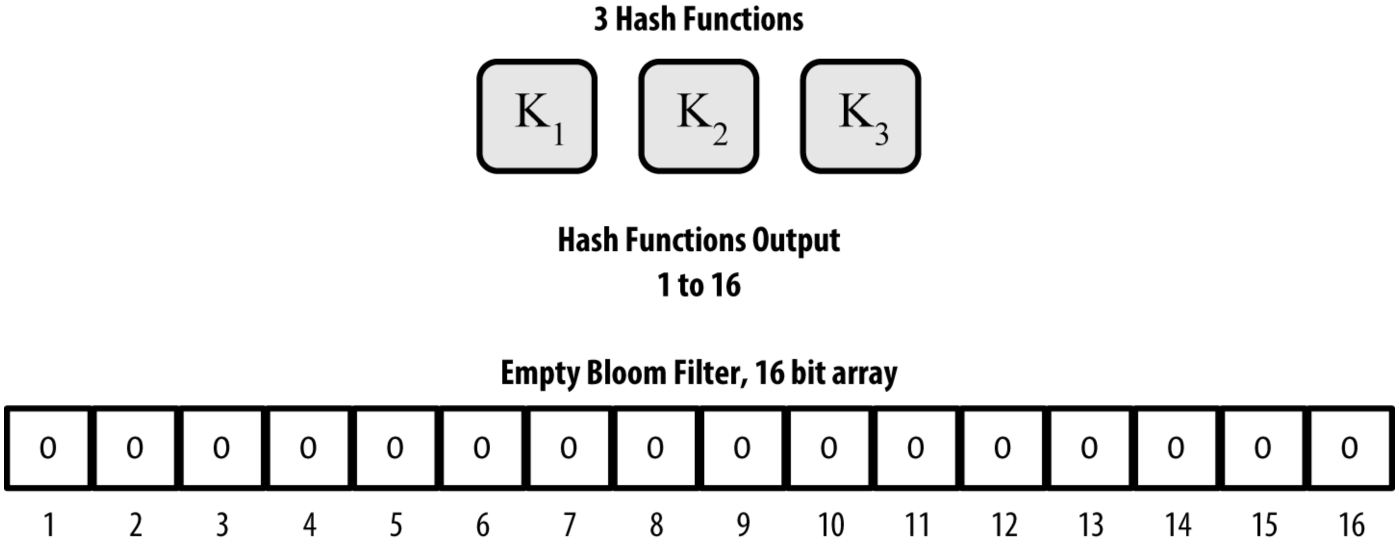
An SPV node:

- Makes a connection using a mask (in the form of a Bloom filter)
- Idea: receive the certificate for the transactions that match the filter

Objective of this class:

- "Install" a Bloom filter when connecting to a full node
- Receive Merkle proof for transactions we are interested in
- Basically, run an SPV node

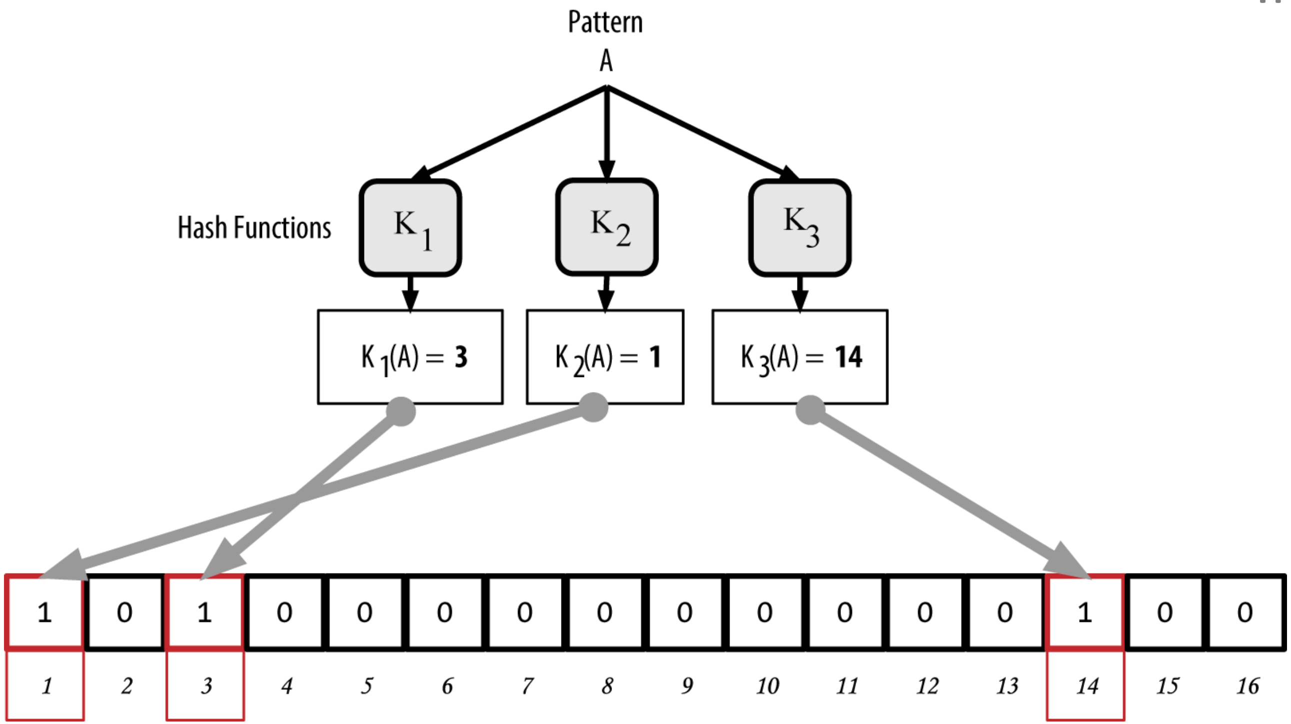
Bloom filters



https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch08.asciidoc

Bloom filters

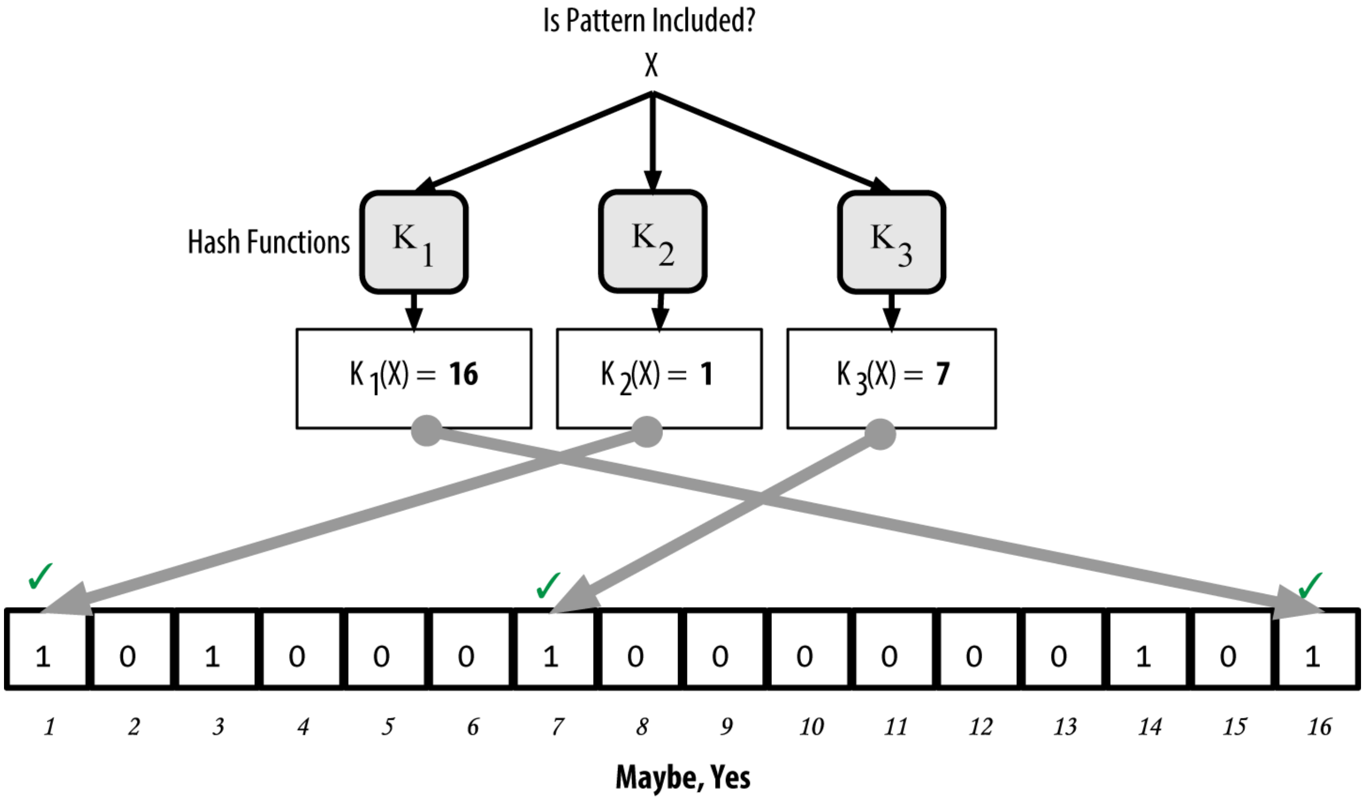
Insertion



https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch08.asciidoc

Bloom filters

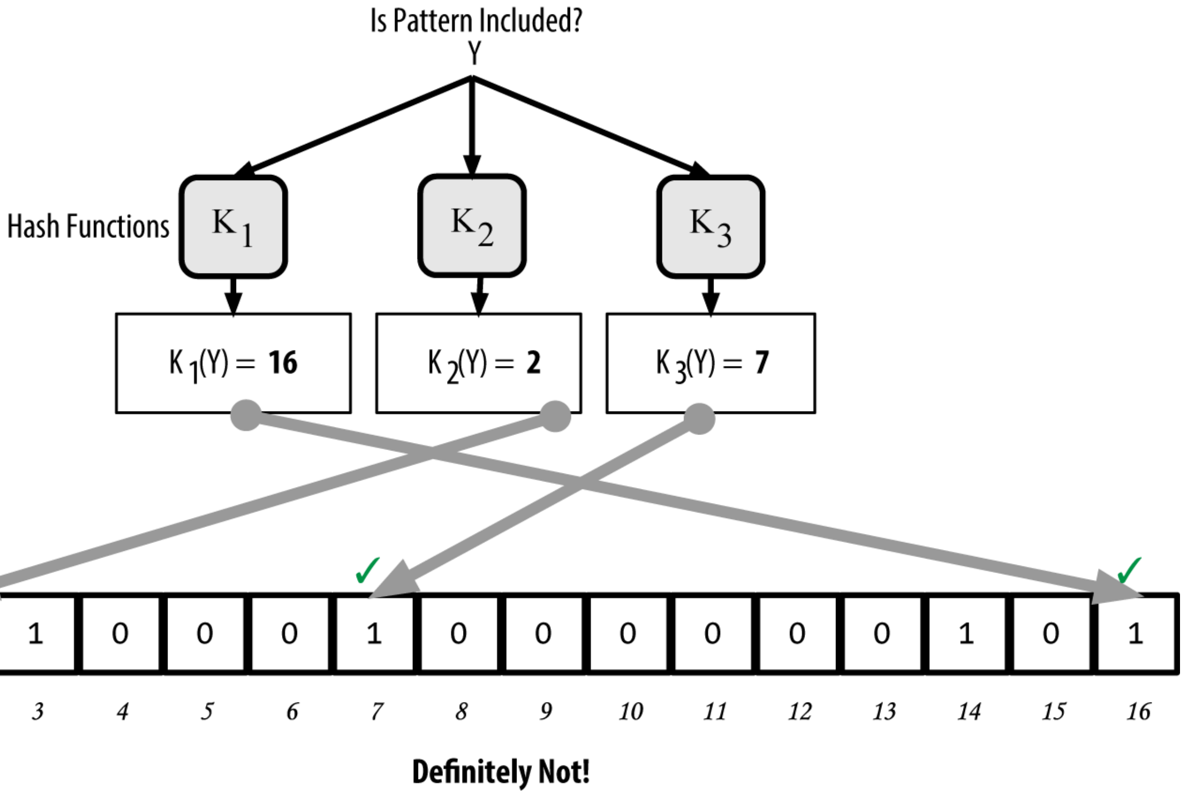
Search



https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch08.asciidoc

Bloom filters

Search



https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch08.asciidoc

Bloom filters

Implementation

```python
BIP37_CONSTANT = 0xfba4c795


class BloomFilter:

    def __init__(self, size, function_count, tweak):
        self.size = size
        self.bit_field = [0] * (size * 8)
        self.function_count = function_count
        self.tweak = tweak
```

bloomfilter.py

Bloom filters

Implementation

Size in bytes

```
BIP37_CONSTANT = 0xfba4c


class BloomFilter:

    def __init__(self, size, function_count, tweak):
        self.size = size
        self.bit_field = [0] * (size * 8)
        self.function_count = function_count
        self.tweak = tweak
```

bloomfilter.py

Bloom filters

Implementation

Size of the bit field

```
BIP37_CONSTANT = 0xfba4c

class BloomFilter:

    def __init__(self, size, function_count, tweak):
        self.size = size
        self.bit_field = [0] * (size * 8)
        self.function_count = function_count
        self.tweak = tweak
```

bloomfilter.py

Bloom filters

Implementation

How many hash functions
will we use

```python
BIP37_CONSTANT = 0xfba4c


class BloomFilter:

    def __init__(self, size, function_count, tweak):
        self.size = size
        self.bit_field = [0] * (size * 8)
        self.function_count = function_count
        self.tweak = tweak
```

bloomfilter.py

Bloom filters

Implementation

Parameter of the hash function

```python
BIP37_CONSTANT = 0xfba4c


class BloomFilter:

    def __init__(self, s  e, function_count, tweak):
        self.size = siz
        self.bit_field = [0] * (size * 8)
        self.function_count = function_count
        self.tweak = tweak
```

bloomfilter.py

Bloom filters

Implementation

Hash function for Bloom filters:

- We always use ***murmur3***
- A function that is not cryptographically safe
- But is super quick and has a decent data distribution

murmur3 takes a seed:

- i * 0xfba4c795 + tweak

Bloom filters

Implementation

Hash function for Bloom filters:

- We always use ***murmur3***
- A function that is not cryptographically safe
- But is super quick and has a decent data distribution

murmur3 takes a seed:

- i * 0xfba4c795 + tweak

i = 0 first function
i = 1 second function
….

Bloom filters

Implementation

Hash function for Bloom filters:

- We always use ***murmur3***
- A function that is not cryptographically safe
- But is super quick and has a decent data distribution

murmur3 takes a seed:

- i * 0xfba4c795 + tweak

BIP 37 constant

Bloom filters

Implementation

Hash function for Bloom filters:

- We always use ***murmur3***
- A function that is not cryptographically safe
- But is super quick and has a decent data distribution

murmur3 takes a seed:

- i * 0xfba4c795 + tweak

A bit of entropy

Bloom filters

murmur3

```python
def murmur3(data, seed=0):
    c1 = 0xcc9e2d51
    c2 = 0x1b873593
    length = len(data)
    h1 = seed
    roundedEnd = (length & 0xfffffffc)  # round down to 4 byte block
    for i in range(0, roundedEnd, 4):
        # little endian load order
        k1 = (data[i] & 0xff) | ((data[i + 1] & 0xff) << 8) | \
            ((data[i + 2] & 0xff) << 16) | (data[i + 3] << 24)
        k1 *= c1
        k1 = (k1 << 15) | ((k1 & 0xffffffff) >> 17)  # ROTL32(k1,15)
        k1 *= c2
        h1 ^= k1
        h1 = (h1 << 13) | ((h1 & 0xffffffff) >> 19)  # ROTL32(h1,13)
        h1 = h1 * 5 + 0xe6546b64

    ...
```

Implemented in helper.py

Bloom filters

Inserting data into the fulter

```python
BIP37_CONSTANT = 0xfba4c795


class BloomFilter:

    def add(self, item):
        '''Add an item to the filter'''
        # iterate self.function_count number of times
        for i in range(self.function_count):
            # BIP0037 spec seed is i*BIP37_CONSTANT + self.tweak
            seed = i * BIP37_CONSTANT + self.tweak
            # get the murmur3 hash given that seed
            h = murmur3(item, seed=seed)
            # set the bit at the hash mod the bitfield size (self.size*8)
            bit = h % (self.size * 8)
            # set the bit field at bit to be 1
            self.bit_field[bit] = 1
```

A message on the Bitcoin network

What does a message look like?

```
f9beb4d976657273696f6e0000000000650000005f1a69d2721101000100000000000000bc8f5e540
0000000010000000000000000000000000000000000000000ffffc61b6409208d010000000000000000
00000000000000ffffcb0071c0208d128035cbc97953f80f2f5361746f7368693a302e392e332fcf0
5050001
```

```
- f9beb4d9 - network magic (always 0xf9beb4d9 for mainnet)
- 76657273696f6e0000000000 - command, 12 bytes, human-readable
- 65000000 - payload length, 4 bytes, little-endian
- 5f1a69d2 - payload checksum, first 4 bytes of hash256 of the
             payload
- 7211...01 - payload
```

https://github.com/jimmysong/programmingbitcoin/blob/master/ch10.asciidoc

A message on the Bitcoin network

What does a message look like?

```
f9beb4d976657273696f6e0000000000650000005f
0000000010000000000000000000000000000000ffff                        000000
000000000000ffffcb0071c0208d128035cbc97953f8    1746f7368693a302e392e332fcf0
5050001
```

b'filterload'

```
- f9beb4d9 - network magic (always 0xf9beb4d9 for mainnet)
- 76657273696f6e0000000000 - command, 12 bytes, human-readable
- 65000000 - payload length, 4 bytes, little-endian
- 5f1a69d2 - payload checksum, first 4 bytes of hash256 of the
             payload
- 7211...01 - payload
```

https://github.com/jimmysong/programmingbitcoin/blob/master/ch10.asciidoc

A message on the Bitcoin network

What does a message look like?

Serialization of the filter content

```
f9beb4d976657273696f6e00000000006500000
00000000100000000000000000000000000000000000 000000
00000000000000ffffcb0071c0208d128035cbc97953f80f2 6f7368693a302e392e332fcf0
5050001
```

- f9beb4d9 - network magic (alw 0xf9beb4d9 for mainnet)
- 76657273696f6e0000000000 - mand, 12 bytes, human-readable
- 65000000 - payload leng , 4 bytes, little-endian
- 5f1a69d2 - payload c ecksum, first 4 bytes of hash256 of the payload
- 7211...01 - payload

https://github.com/jimmysong/programmingbitcoin/blob/master/ch10.asciidoc

Bloom filters

Payload for filterload

```
0a4000600a080000010940050000006300000000

- 0a4000600a080000010940 - Bit field, variable field
- 05000000 - Hash count, 4 bytes, little-endian
- 63000000 - Tweak, 4 bytes, little-endian
- 00 - Matched item flag
```

https://github.com/jimmysong/programmingbitcoin/blob/master/ch12.asciidoc

Bloom filters

Payload for filterload

Bitfield of the filter
Format: varint(length) + bits_in_bytes

`0a4000600a0800000109400` `00006300000000`

- `0a4000600a080000010940` - Bit field, variable field
- `05000000` - Hash count, 4 bytes, little-endian
- `63000000` - Tweak, 4 bytes, little-endian
- `00` - Matched item flag

https://github.com/jimmysong/programmingbitcoin/blob/master/ch12.asciidoc

Bloom filters

Payload for filterload

Number of hash functions
used for this filter

0a4000600a080000 0050000006300000000

- 0a4000600a0800000 0940 - Bit field, variable field
- 05000000 - Hash count, 4 bytes, little-endian
- 63000000 - Tweak, 4 bytes, little-endian
- 00 - Matched item flag

https://github.com/jimmysong/programmingbitcoin/blob/master/ch12.asciidoc

Bloom filters

Payload for filterload

Tweak used for this filter

0a4000600a0800[illegible]940050000006300000000

- 0a4000600a0800[illegible]010940 - Bit field, variable field
- 05000000 - Has[illegible] count, 4 bytes, little-endian
- 63000000 - Tweak, 4 bytes, little-endian
- 00 - Matched item flag

https://github.com/jimmysong/programmingbitcoin/blob/master/ch12.asciidoc

Bloom filters

Payload for filterload

Flag specifiying how data is inserted
into the filter

0a4000600a080 0940050000006300000000

- 0a4000600a0 000010940 - Bit field, variable field
- 05000000 - ash count, 4 bytes, little-endian
- 63000000 - Tweak, 4 bytes, little-endian
- 00 - Matched item flag

https://github.com/jimmysong/programmingbitcoin/blob/master/ch12.asciidoc

Bloom filters

Inserting data

```
BIP37_CONSTANT = 0xfba4c795


class BloomFilter:
```

```
    def filter_bytes(self):
        return bit_field_to_bytes(self.bit_field)

    def filterload(self, flag=1):
        '''Return the filterload message'''
        # start the payload with the size of the filter in bytes
        payload = encode_varint(self.size)
        # next add the bit field using self.filter_bytes()
        payload += self.filter_bytes()
        # function count is 4 bytes little endian
        payload += int_to_little_endian(self.function_count, 4)
        # tweak is 4 bytes little endian
        payload += int_to_little_endian(self.tweak, 4)
        # flag is 1 byte little endian
        payload += int_to_little_endian(flag, 1)
        # return a GenericMessage whose command is b'filterload'
        # and payload is what we've calculated
        return GenericMessage(b'filterload', payload)
```

Bloom filters

Inserting data

```python
def bit_field_to_bytes(bit_field):
    if len(bit_field) % 8 != 0:
        raise RuntimeError('bit_field does not have a length that is divisible by 8')
    result = bytearray(len(bit_field) // 8)
    for i, bit in enumerate(bit_field):
        byte_index, bit_index = divmod(i, 8)
        if bit:
            result[byte_index] |= 1 << bit_index
    return bytes(result)


def bytes_to_bit_field(some_bytes):
    flag_bits = []
    # iterate over each byte of flags
    for byte in some_bytes:
        # iterate over each bit, right-to-left
        for _ in range(8):
            # add the current bit (byte & 1)
            flag_bits.append(byte & 1)
            # rightshift the byte 1
            byte >>= 1
    return flag_bits
```

helper.py

Bloom filters

As used in a connection

```
# last block we know:
last_block_hex = '00000000000002e5fc775089469c567efc54879bd23172edcdda29f9f0242342'
# stuff we're looking for (it's in block 25):
address = 'n3jKhCmVjvaVgg8C5P7E48fdRkQAAvf7Wc'
h160 = decode_base58(address)


# Establish a connection to a testnet node#
node = SimpleNode('testnet.programmingbitcoin.com', testnet=True, logging=False)
# Define our bloom filter
bf = BloomFilter(size=30, function_count=5, tweak=90210)
# Put the data into the filter
bf.add(h160)
# Handshake and load the filter onto the connection
node.handshake()
node.send(bf.filterload())
```

spv.py

Recall

Objective of this class:

- "Install" a Bloom filter when connecting to a full node ✓
- **Receive Merkle proof for transactions we are interested in**

Basically, we still need to ask for them. Bloom filter determines what we get!

Asking for Merkle proof

getdata

A *getdata* message with the following payload:

```
020300000030eb2540c41025690160a1014c577061596e32e426b712c7ca000
00000000000030000001049847939585b0652fba793661c361223446b6fc410
89b8be00000000000000
```

```
- 02 - Number of data items
- 03000000 - Type of data item (tx, block, filtered
        block, compact block), little-endian
- 30...00 - Hash identifier
```

https://github.com/jimmysong/programmingbitcoin/blob/master/ch12.asciidoc

Asking for Merkle proof

getdata

How many?
Format: varint

A *getdata* message wit

```
020300000030eb2540c41025690160a1014c577061596e32e426b712c7ca000
00000000000030000001049847939585b0652fba793661c361223446b6fc410
89b8be00000000000000

- 02 - Number of data items
- 03000000 - Type of data item (tx, block, filtered
        block, compact block), little-endian
- 30...00 - Hash identifier
```

https://github.com/jimmysong/programmingbitcoin/blob/master/ch12.asciidoc

Asking for Merkle proof

getdata

Type of data
3 = MerkleProof
(for the things matching the BF)

A *getdata* message wit the

```
020300000030eb2540c41025690160a1014c577061596e32e426b712c7ca000
000000000000300000001049847939585b0652fba793661c361223446b6fc410
89b8be00000000000000
```

```
- 02 - Number of data items
- 03000000 - Type of data item (tx, block, filtered
        block, compact block), little-endian
- 30...00 - Hash identifier
```

https://github.com/jimmysong/programmingbitcoin/blob/master/ch12.asciidoc

Asking for Merkle proof

getdata

Hash of the thing we asked for

A *getdata* message wit the follow

```
020300000030eb2540c41025690160a1014c577061596e32e426b712c7ca000
0000000000030000001049847939585b0652fba793661c361223446b6fc410
89b8be00000000000000
```

```
- 02 - Number of data items
- 03000000 - Type of data item (tx, block, filtered
        block, compact block), little-endian
- 30...00 - Hash identifier
```

https://github.com/jimmysong/programmingbitcoin/blob/master/ch12.asciidoc

Asking for Merkle proof

getdata

```
# Get block headers (2000 starting from last_block_hex)
start_block = bytes.fromhex(last_block_hex)
getheaders = GetHeadersMessage(start_block=start_block)
node.send(getheaders)
headers = node.wait_for(HeadersMessage)

# Load a get data message with this stuff
getdata = GetDataMessage()
for b in headers.blocks:
    if not b.check_pow():
        raise RuntimeError('proof of work is invalid')
    getdata.add_data(FILTERED_BLOCK_DATA_TYPE, b.hash())

# Ask for data in these headers
node.send(getdata)

# The node replying to this message will send:
# 1. A MerkleBlock with:
#        - A Merkle Proof when a tx matches the filter
#        - With empty Merkle Proof otherwise (just the root)
# 2. A Tx message if any of the Txs in the block matches the filter
```

spv.py

Asking for Merkle proof

getdata

```python
# Get block headers (2000 starting from last_block_hex)
start_block = bytes.fromhex(last_block_hex)
getheaders = GetHeadersMessage(start_block=start_block)
node.send(getheaders)
headers = node.wait_for(HeadersMessage)

# Load a get data message with this stuff
getdata = GetDataMessage()
for b in headers.blocks:
    if not b.check_pow():
        raise RuntimeError('proof of work is invalid')
    getdata.add_data(FILTERED_BLOCK_DATA_TYPE, b.hash())

# Ask for data in these headers
node.send(getdata)

# The node replying to this message will send:
# 1. A MerkleBlock with:
#       - A Merkle Proof when a tx matches the filter
#       - With empty Merkle Proof otherwise (just the root)
# 2. A Tx message if any of the Txs in the block matches the filter
```

What will I get as a reply?

merkleblock

spv.py

Merkle proof

merkleblock

```
00000020df3b053dc46f162a9b00c7f0d5124e2676d47bbe7c5d0793a500000000000000ef445fef2
ed495c275892206ca533e7411907971013ab83e3b47bd0d692d14d4dc7c835b67d8001ac157e670bf
0d00000aba412a0d1480e370173072c9562becffe87aa661c1e4a6dbc305d38ec5dc088a7cf92e645
8aca7b32edae818f9c2c98c37e06bf72ae0ce80649a38655ee1e27d34d9421d940b16732f24b94023
e9d572a7f9ab8023434a4feb532d2adfc8c2c2158785d1bd04eb99df2e86c54bc13e139862897217
400def5d72c280222c4cbaee7261831e1550dbb8fa82853e9fe506fc5fda3f7b919d8fe74b6282f927
63cef8e625f977af7c8619c32a369b832bc2d051ecd9c73c51e76370ceabd4f25097c256597fa898d
404ed53425de608ac6bfe426f6e2bb457f1c554866eb69dcb8d6bf6f880e9a59b3cd053e6c7060eea
caacf4dac6697dac20e4bd3f38a2ea2543d1ab7953e3430790a9f81e1c67f5b58c825acf46bd02848
384eebe9af917274cdfbb1a28a5d58a23a17977def0de10d644258d9c54f886d47d293a411cb62261
03b55635
```

- 00000020 - version, 4 bytes, LE
- df3b...00 - previous block, 32 bytes, LE
- ef44...d4 - Merkle root, 32 bytes, LE
- dc7c835b - timestamp, 4 bytes, LE
- 67d8001a - bits, 4 bytes
- c157e670 - nonce, 4 bytes
- bf0d0000 - number of total transactions, 4 bytes, LE
- 0a - number of hashes, varint
- ba41...61 - hashes, 32 bytes * number of hashes
- 03b55635 - flag bits

https://github.com/jimmysong/programmingbitcoin/blob/master/ch10.asciidoc

Merkle proof

merkleblock

```
00000020df3b053dc46f162a9b00c7f0d5124e2676d47bbe7c5d0793a500000000000000ef445fef2
ed495c275892206ca533e7411907971013ab83e3b47bd0d692d14d4dc7c835b67d8001ac157e670bf
0d00000aba412a0d1480e370173072c9562becffe87aa661c1e4a6dbc305d38ec5dc088a7cf92e645
8aca7b32edae818f9c2c98c37e06bf72ae0ce80649a38655ee1e27d34d9421d940b16732f24b94023
e9d572a7f9ab8023434a4feb532d2adfc8c2c2158785d1bd04eb99df2e86c54bc13e139862897217400def5d72c280222c4cbaee7261831e1550dbb8fa82853e9fe506fc5fda3f7b919d8fe74b6282f927
63cef8e625f977af7c8619c32a369b832bc2d051ecd9c73c51e76370ceabd4f25097c256597fa898d
404ed53425de608ac6bfe426f6e2bb457f1c554866eb69dcb8d6bf6f880e9a59b3cd053e6c7060eea
caacf4dac6697dac20e4bd3f38a2ea2543d1ab7953e3430790a9f81e1c67f5b58c825acf46bd02848
384eebe9af917274cdfbb1a28a5d58a23a17977def0de10d644258d9c54f886d47d293a411cb62261
03b55635
```

Block header
(but with the real tx nr.)

```
- 00000020 - version, 4 bytes, LE
- df3b...00 - previous block, 32 bytes, LE
- ef44...d4 - Merkle root, 32 bytes, LE
- dc7c835b - timestamp, 4 bytes, LE
- 67d8001a - bits, 4 bytes
- c157e670 - nonce, 4 bytes
- bf0d0000 - number of total transactions, 4 bytes, LE
- 0a - number of hashes, varint
- ba41...61 - hashes, 32 bytes * number of hashes
- 03b55635 - flag bits
```

https://github.com/jimmysong/programmingbitcoin/blob/master/ch10.asciidoc

Merkle proof

merkleblock

```
00000020df3b053dc46f162a9b00c7f0d5124e2676d47bbe7c5d0793a500000000000000ef445fef2
ed495c275892206ca533e7411907971013ab83e3b47bd0d692d14d4dc7c835b67d8001ac157e670bf
0d00000aba412a0d1480e370173072c9562becffe87aa661c1e4a6dbc305d38ec5dc088a7cf92e645
8aca7b32edae818f9c2c98c37e06bf72ae0ce80649a38655ee1e27d34d9421d940b16732f24b94023
e9d572a7f9ab8023434a4feb532d2adfc8c2c2158785d1bd04eb99df2e86c54bc13e139862897217400def5d72c280222c4cbaee7261831e1550dbb8fa82853e9fe506fc5fda3f7b919d8fe74b6282f927
63cef8e625f977af7c8619c32a369b832bc2d051ecd9c73c51e76370ceabd4f25097c256597fa898d
404ed53425de608ac6bfe426f6e2bb457f1c554866eb69dcb8d6bf6f880e9a59b3cd053e6c7060eea
caacf4dac6697dac20e4bd3f38a2ea2543d1ab7953e3430790a9f81e1c67f5b58c825acf46bd02848
384eebe9af917274cdfbb1a28a5d58a23a17977def0de10d644258d9c54f886d47d293a411cb62261
03b55635
```

- 00000020 - version, 4 bytes, LE
- df3b...00 - previous block, 32 bytes, LE
- ef44...d4 - Merkle root, 32 bytes, LE
- dc7c835b - timestamp, 4 bytes, LE
- 67d8001a - bits, 4 bytes
- c157e670 - nonce, 4 bytes
- bf0d0000 - number of total transactions, 4 bytes, LE
- 0a - number of hashes, varint
- ba41...61 - hashes, 32 bytes * number of hashes
- 03b55635 - flag bits

Merkle proof

https://github.com/jimmysong/programmingbitcoin/blob/master/ch10.asciidoc

Merkle proof

merkleblock

```python
class MerkleBlock:
    command = b'merkleblock'

    def __init__(self, version, prev_block, merkle_root, timestamp, bits, nonce, total, hashes, flags):
        self.version = version
        self.prev_block = prev_block
        self.merkle_root = merkle_root
        self.timestamp = timestamp
        self.bits = bits
        self.nonce = nonce
        self.total = total
        self.hashes = hashes
        self.flags = flags
```

merkleblock.py

Merkle proof

merkleblock

It is a message
(as all things in BTC)

```python
class MerkleBlock:
    command = b'merkleblock'

    def __init__(self, version, prev_block, merkle_root, timestamp, bits, nonce, total, hashes, flags):
        self.version = version
        self.prev_block = prev_block
        self.merkle_root = merkle_root
        self.timestamp = timestamp
        self.bits = bits
        self.nonce = nonce
        self.total = total
        self.hashes = hashes
        self.flags = flags
```

merkleblock.py

Merkle proof

merkleblock

```python
def parse(cls, s):
    '''Takes a byte stream and parses a merkle block. Returns a Merkle Block object'''
    # version - 4 bytes, Little-Endian integer
    version = little_endian_to_int(s.read(4))
    # prev_block - 32 bytes, Little-Endian (use [::-1])
    prev_block = s.read(32)[::-1]
    # merkle_root - 32 bytes, Little-Endian (use [::-1])
    merkle_root = s.read(32)[::-1]
    # timestamp - 4 bytes, Little-Endian integer
    timestamp = little_endian_to_int(s.read(4))
    # bits - 4 bytes
    bits = s.read(4)
    # nonce - 4 bytes
    nonce = s.read(4)
    # total transactions in block - 4 bytes, Little-Endian integer
    total = little_endian_to_int(s.read(4))
    # number of transaction hashes - varint
    num_hashes = read_varint(s)
    # each transaction is 32 bytes, Little-Endian
    hashes = []
    for _ in range(num_hashes):
        hashes.append(s.read(32)[::-1])
    # length of flags field - varint
    flags_length = read_varint(s)
    # read the flags field
    flags = s.read(flags_length)
    # initialize class
    return cls(version, prev_block, merkle_root, timestamp, bits, nonce,
               total, hashes, flags)
```

merkleblock.py

Merkle proof

merkleblock

For validating the proof

```
class MerkleBlock:
    command = b'merkleblock'

    def is_valid(self):
        '''Verifies whether the merkle tree information validates to the merkle root'''
        # convert the flags field to a bit field
        flag_bits = bytes_to_bit_field(self.flags)
        # reverse self.hashes for the merkle root calculation
        hashes = [h[::-1] for h in self.hashes]
        # initialize the merkle tree
        merkle_tree = MerkleTree(self.total)
        # populate the tree with flag bits and hashes
        merkle_tree.populate_tree(flag_bits, hashes)
        # check if the computed root reversed is the same as the merkle root
        return merkle_tree.root()[::-1] == self.merkle_root
```

merkleblock.py

Merkle proof

merkleblock

For validating the proof

```python
class MerkleBlock:
    command = b'merkleblock'

    def is_valid(self):
        '''Verifies whether the merkle tree information validates to th
        # convert the flags field to a bit field
        flag_bits = bytes_to_bit_field(self.flags)
        # reverse self.hashes for the merkle root calculation
        hashes = [h[::-1] for h in self.hashes]
        # initialize the merkle tree
        merkle_tree = MerkleTree(self.total)
        # populate the tree with flag bits and hashes
        merkle_tree.populate_tree(flag_bits, hashes)
        # check if the computed root reversed is the same as the merkle root
        return merkle_tree.root()[::-1] == self.merkle_root
```

What is this?

Week 3
(flag bits!!!)

merkleblock.py

Merkle proof

merkleblock

```
class MerkleTree:

    def __init__(self, total):
        self.total = total
        # compute max depth math.ceil(math.log(self.total, 2))
        self.max_depth = math.ceil(math.log(self.total, 2))
        # initialize the nodes property to hold the actual tree
        self.nodes = []
        # loop over the number of levels (max_depth+1)
        for depth in range(self.max_depth + 1):
            # the number of items at this depth is
            # math.ceil(self.total / 2**(self.max_depth - depth))
            num_items = math.ceil(self.total / 2**(self.max_depth - depth))
            # create this level's hashes list with the right number of items
            level_hashes = [None] * num_items
            # append this level's hashes to the merkle tree
            self.nodes.append(level_hashes)
        # set the pointer to the root (depth=0, index=0)
        self.current_depth = 0
        self.current_index = 0
```

merkleblock.py

Merkle proof

merkleblock

```python
class MerkleTree:

    def populate_tree(self, flag_bits, hashes):
        # populate until we have the root
        while self.root() is None:
            # if we are a leaf, we know this position's hash
            if self.is_leaf():
                # get the next bit from flag_bits: flag_bits.pop(0)
                flag_bits.pop(0)
                # set the current node in the merkle tree to the next has
                self.set_current_node(hashes.pop(0))
                # go up a level
                self.up()
            else:
                # get the left hash
                left_hash = self.get_left_node()
                # if we don't have the left hash
                if left_hash is None:
                    # if the next flag bit is 0, the next hash is our current node
                    if flag_bits.pop(0) == 0:
                        # set the current node to be the next hash
                        self.set_current_node(hashes.pop(0))
                        # sub-tree doesn't need calculation, go up
                        self.up()
                    else:
```

Week 3
…

merkleblock.py

Merkle proof

transactions

With getadata for MerkleProofs I can also get a transaction:

- Is our implementation of Tx good enough to send a message on the network?

Of course!

Each element is parsed/serialized as it is communicated by the network

We only need to add a single word to our implementation of tx.py

Merkle proof

Transactions – before

```python
class Tx:

    def __init__(self, version, tx_ins, tx_outs, locktime, testnet=False):
        self.version = version
        self.tx_ins = tx_ins
        self.tx_outs = tx_outs
        self.locktime = locktime
        self.testnet = testnet
```

tx.py

Merkle proof

Transactions – as messages

It is really that easy!

```python
class Tx:
    command = b'tx'

    def __init__(self, version, tx_ins, tx_outs, locktime, testnet=False):
        self.version = version
        self.tx_ins = tx_ins
        self.tx_outs = tx_outs
        self.locktime = locktime
        self.testnet = testnet
```

tx.py

SPV node

filterload

```python
# last block we know:
last_block_hex = '00000000000002e5fc775089469c567efc54879bd23172edcdda29f9f0242342'
# stuff we're looking for (it's in block 25):
address = 'n3jKhCmVjvaVgg8C5P7E48fdRkQAAvf7Wc'
h160 = decode_base58(address)


# Establish a connection to a testnet node#
node = SimpleNode('testnet.programmingbitcoin.com', testnet=True, logging=False)
# Define our bloom filter
bf = BloomFilter(size=30, function_count=5, tweak=90210)
# Put the data into the filter
bf.add(h160)
# Handshake and load the filter onto the connection
node.handshake()
node.send(bf.filterload())
```

spv.py

SPV node

MerkleProofs

```
# Get block headers (2000 starting from last_block_hex)
start_block = bytes.fromhex(last_block_hex)
getheaders = GetHeadersMessage(start_block=start_block)
node.send(getheaders)
headers = node.wait_for(HeadersMessage)

# Load a get data message with this stuff
getdata = GetDataMessage()
for b in headers.blocks:
    if not b.check_pow():
        raise RuntimeError('proof of work is invalid')
    getdata.add_data(FILTERED_BLOCK_DATA_TYPE, b.hash())

# Ask for data in these headers
node.send(getdata)

# The node replying to this message will send:
# 1. A MerkleBlock with:
#        - A Merkle Proof when a tx matches the filter
#        - With empty Merkle Proof otherwise (just the root)
# 2. A Tx message if any of the Txs in the block matches the filter
```

spv.py

SPV node

MerkleProofs

```python
# Get block headers (2000 starting from last_block_hex)
start_block = bytes.fromhex(last_block_hex)
getheaders = GetHeadersMessage(start_block=start_block)
node.send(getheaders)
headers = node.wait_for(HeadersMessage)

# Load a get data message with this stuff
getdata = GetDataMessage()
for b in headers.blocks:
    if not b.check_pow():
        raise RuntimeError('proof of work is invalid')
    getdata.add_data(FILTERED_BLOCK_DATA_TYPE, b.hash())

# Ask for data in these headers
node.send(getdata)

# The node replying to this message will send:
# 1. A MerkleBlock with:
#        - A Merkle Proof when a tx matches the filter
#        - With empty Merkle Proof otherwise (just the root)
# 2. A Tx message if any of the Txs in the block matches the filter
```

I’m askin for txs in these blocks that match the Bloom filter!

spv.py

SPV node

Answers

```
# Wait for the data; this is a bit poorly implemented
# We have 2000 hashes above to run through, so we'll exhaust them
# The breaking conditions need work, but this is just to demo ;-)
# Namely, I know that we will receive 2003 messages; should probably wait for s
j = 2003
while j>0:
    message = node.wait_for(MerkleBlock, Tx)
    j = j - 1
    # A mekleblock message that matches the filter will send the proof as well
    # The one that does not will just send a single hash proof (the root)
    if message.command == b'merkleblock':
        #print('block:', message.hash().hex())
        #if (len(message.hashes)>1):
        #for x in message.hashes:
        #   print(x.hex()[::-1])
        if not message.is_valid():
            raise RuntimeError('invalid merkle proof')
    # Here we check if output matches our address
    # In this case we print it out
    else:
        for i, tx_out in enumerate(message.tx_outs):
            if tx_out.script_pubkey.address(testnet=True) == address:
                print('found: {}:{}'.format(message.id(), i))
```

spv.py

SPV node

Answers

```
# Wait for the data; this is a bit poorly implemented
# We have 2000 hashes above to run through, so we'll exhaust them
# The breaking conditions need work, but this is just to demo ;-)
# Namely, I know that we will receive 2003 messages; should probably wait for s
j = 2003
while j>0:
    message = node.wait_for(MerkleBlock, Tx)
    j = j - 1
    # A mekleblock message that matches the filter will send
    # The one that does not will just send a single hash proof (the r
    if message.command == b'merkleblock':
        #print('block:', message.hash().hex())
        #if (len(message.hashes)>1):
        #for x in message.hashes:
        #   print(x.hex()[::-1])
        if not message.is_valid():
            raise RuntimeError('invalid merkle proof')
    # Here we check if output matches our address
    # In this case we print it out
    else:
        for i, tx_out in enumerate(message.tx_outs):
            if tx_out.script_pubkey.address(testnet=True) == address:
                print('found: {}:{}'.format(message.id(), i))
```

We will receive:
-MerkleBlock
-Tx

spv.py

SPV node

Answers

```python
# Wait for the data; this is a bit poorly implemented
# We have 2000 hashes above to run through, so we'll exhaust them
# The breaking conditions need work, but this is just to demo ;-)
# Namely, I know that we will receive 2003 messages; should probably wait for s
j = 2003
while j>0:
    message = node.wait_for(MerkleBlock, Tx)
    j = j - 1
    # A mekleblock message that matches the filter will send the proo
    # The one that does not will just send a single hash proof (the
    if message.command == b'merkleblock':
        #print('block:', message.hash().hex())
        #if (len(message.hashes)>1):
        #for x in message.hashes:
        #    print(x.hex()[::-1])
        if not message.is_valid():
            raise RuntimeError('invalid merkle proof')
    # Here we check if output matches our address
    # In this case we print it out
    else:
        for i, tx_out in enumerate(message.tx_outs):
            if tx_out.script_pubkey.address(testnet=True) == add
                print('found: {}:{}'.format(message.id(), i))
```

In case of MerklBlock:
We need to validate the proof

If our address is not in this block the proof contains only MerkleRoot

spv.py

SPV node

Answers

```
# Wait for the data; this is a bit poorly implemented
# We have 2000 hashes above to run through, so we'll exhaust them
# The breaking conditions need work, but this is just to demo ;-)
# Namely, I know that we will receive 2003 messages; should probably wait for s
j = 2003
while j>0:
    message = node.wait_for(MerkleBlock, Tx)
    j = j - 1
    # A mekleblock message that matches the filter will send the proof as wel
    # The one that does not will just send a single hash proof (the root
    if message.command == b'merkleblock':
        #print('block:', message.hash().hex())
        #if (len(message.hashes)>1):
        #for x in message.hashes:
        #   print(x.hex()[::-1])
        if not message.is_valid():
            raise RuntimeError('invalid merkle proof')
    # Here we check if output matches our address
    # In this case we print it out
    else:
        for i, tx_out in enumerate(message.tx_outs):
            if tx_out.script_pubkey.address(testnet=True) == address:
                print('found: {}:{}'.format(message.id(), i))
```

In case of Tx

spv.py

SPV node

Answers

```python
# Wait for the data; this is a bit poorly implemented
# We have 2000 hashes above to run through, so we'll exhaust them
# The breaking conditions need work, but this is just to demo ;-)
# Namely, I know that we will receive 2003 messages; should probably wait for s
j = 2003
while j>0:
    message = node.wait_for(MerkleBlock, Tx)
    j = j - 1
    # A mekleblock message that matches the filter will send the
    # The one that does not will just send a single hash proof
    if message.command == b'merkleblock':
        #print('block:', message.hash().hex())
        #if (len(message.hashes)>1):
        #for x in message.hashes:
        #   print(x.hex()[::-1])
        if not message.is_valid():
            raise RuntimeError('invalid merkle proof')
    # Here we check if output matches our address
    # In this case we print it out
    else:
        for i, tx_out in enumerate(message.tx_outs):
            if tx_out.script_pubkey.address(testnet=True) == address:
                print('found: {}:{}'.format(message.id(), i))
```

Searching our address
(Method address
implemented in script.py)

spv.py

SPV node

Answers

```
# Wait for the data; this is a bit poorly implemented
# We have 2000 hashes above to run through, so we'll exhaust them
# The breaking conditions need work, but this is just to demo ;-)
# Namely, I know that we will receive 2003 messages; should probably wait for s
j = 2003
while j>0:
    message = node.wait_for(Mer        Tx)
    j = j - 1
    # A mekleblock message that matches the filter wi
    # The one that does not will just send a single hash proo
    if message.command == b'merkleblock':
        #print('block:', message.hash().hex())
        #if (len(message.hashes)>1):
        #for x in message.hashes:
        #   print(x.hex()[::-1])
        if not message.is_valid():
            raise RuntimeError('invalid merkle proof')
    # Here we check if output matches our address
    # In this case we print it out
    else:
        for i, tx_out in enumerate(message.tx_outs):
            if tx_out.script_pubkey.address(testnet=True) == address:
                print('found: {}:{}'.format(message.id(), i))
```

2000 MerkleProofs
+
3 transactions
(real node has async stuff)
(and a better logic)

spv.py

Bitcoin

Our implementation

To run a node:

- **network.py –** for sending network messages
- **bloomfilter.py –** manage Bloom filters in connections
- **merkleblock.py –** manages Merkle proofs
- **block.py –** manages headers and complete blocks
- **tx.py –** manages transactions
- **script.py –** for processing inputs/outputs of a transaction
- **op.py –** to process commands of Script
- **ecc.py** – for crypto
- **helper.py –** utility functions (coding/decoding)

References

Read:

- Programming Bitcoin, chapters 11,12
- https://en.bitcoin.it/wiki/Protocol_documentation