Red de Bitcoin

Merkle Blocks y Filtros de Bloom

Bitcoin network

Las referencias que realmente necesitan:

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

https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki
(también contiene solución de la tarea 1 ☺)

Bitcoin network

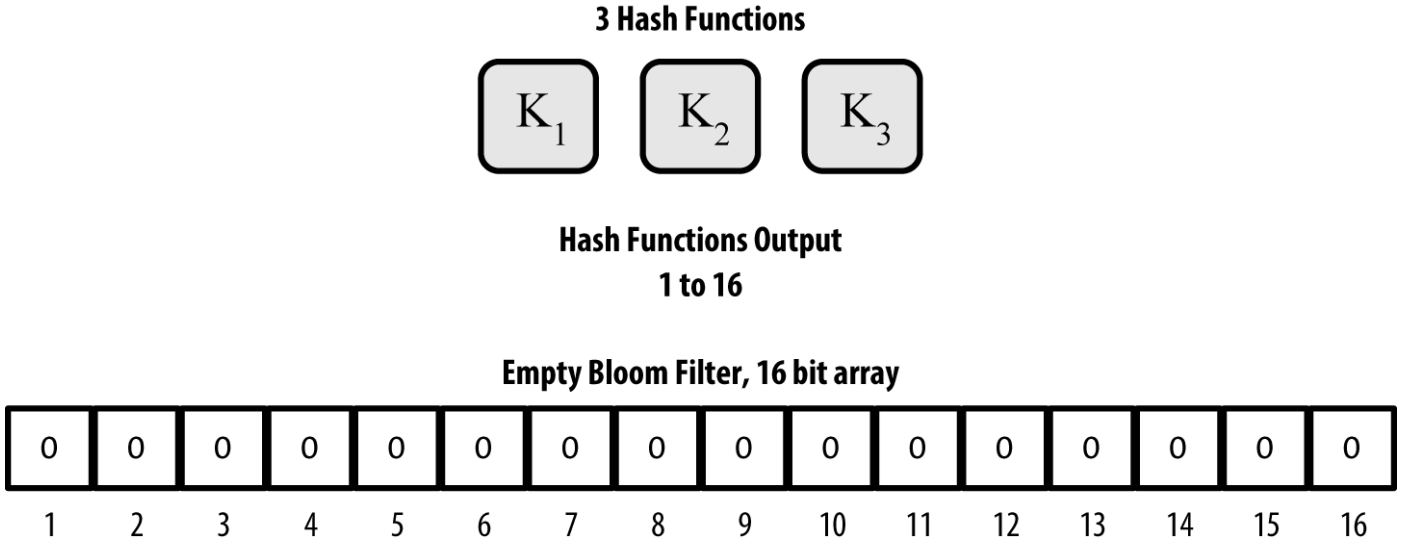
Un nodo SPV:

- Hace conexión con una mascarilla (en forma de Filtro de Bloom)
- Idea: recibir la prueba para las transacciones que pasan por el filtro

Objetivo de esta clase:

- "Instalar" un filtro de Bloom conectándose a un nodo
- Recibir pruebas de Merkle de las transacciones que nos interesan

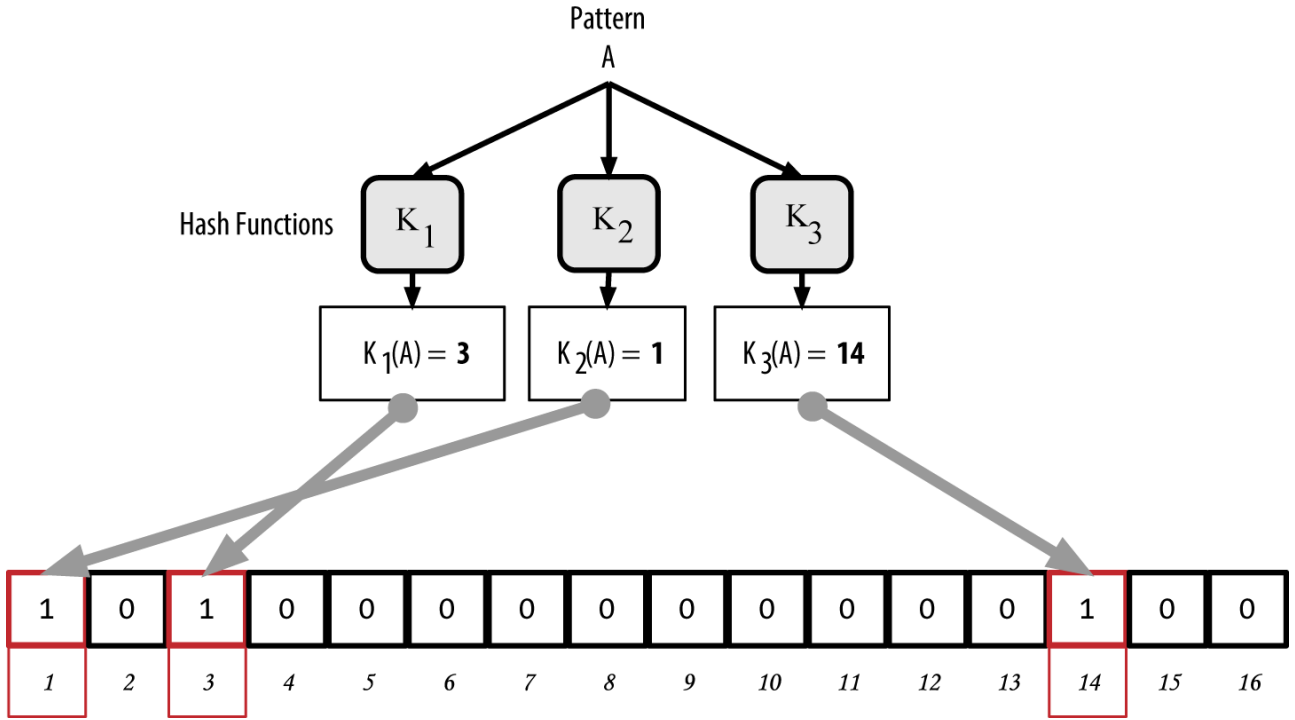
Filtros de Bloom



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

Filtros de Bloom

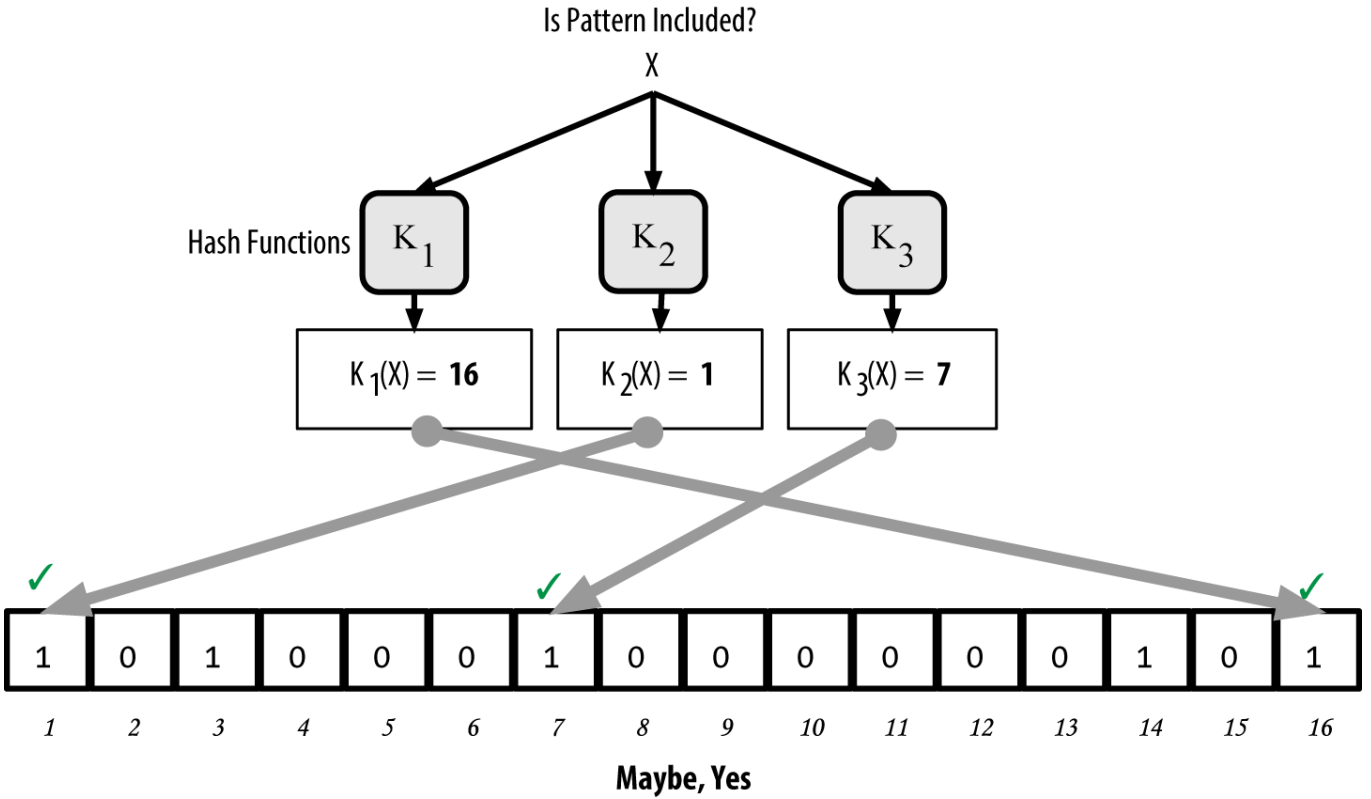
Inserción



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

Filtros de Bloom

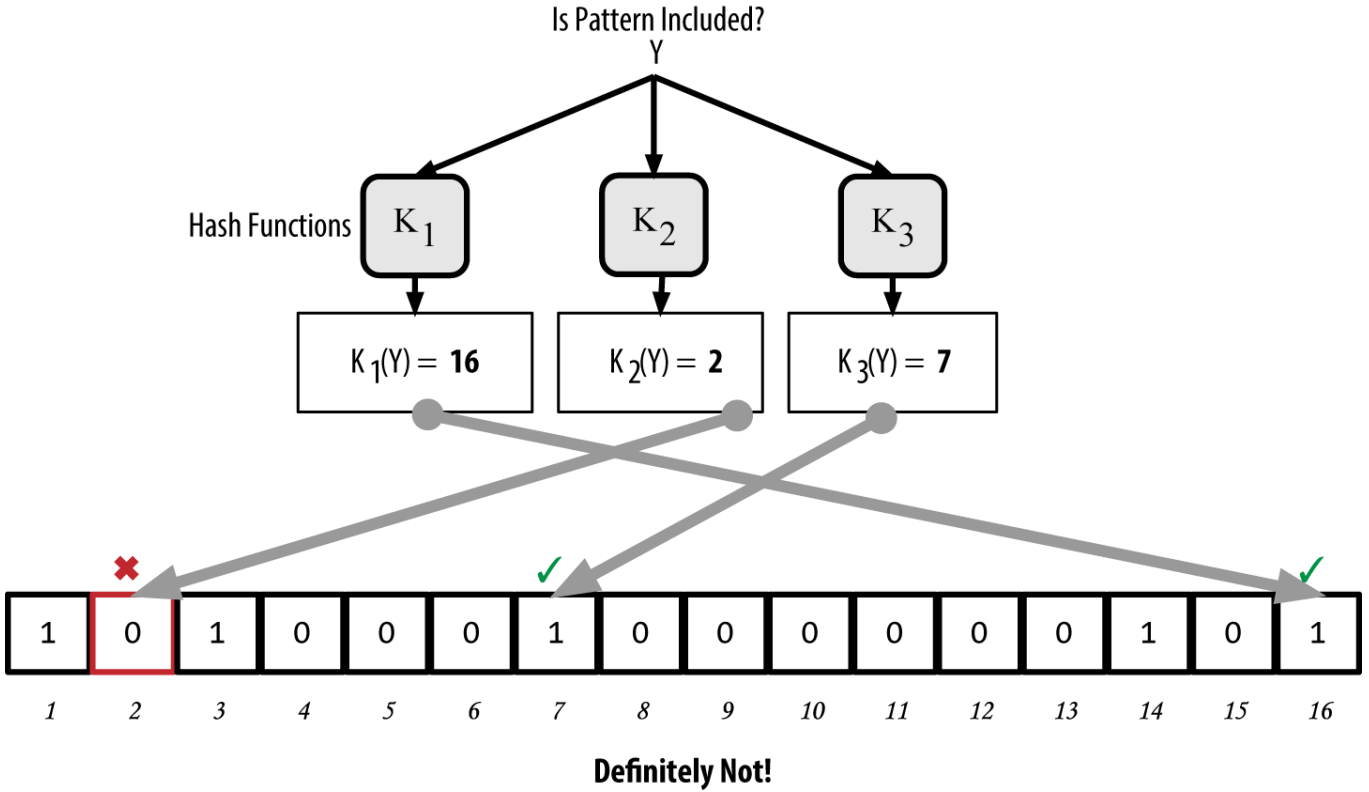
Busqeda



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

Filtros de Bloom

Busqeda



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

Filtros de Bloom

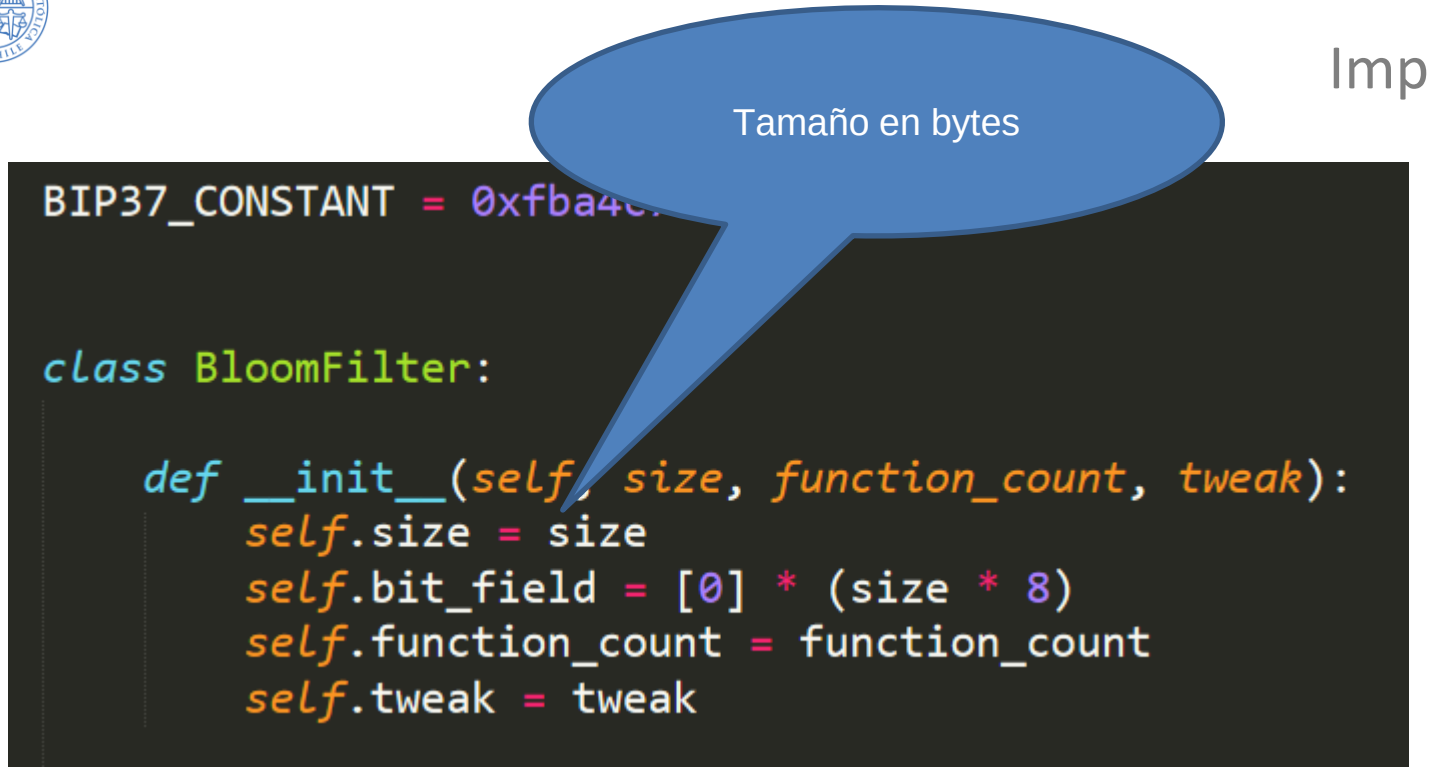
Implementación

```
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

Filtros de Bloom

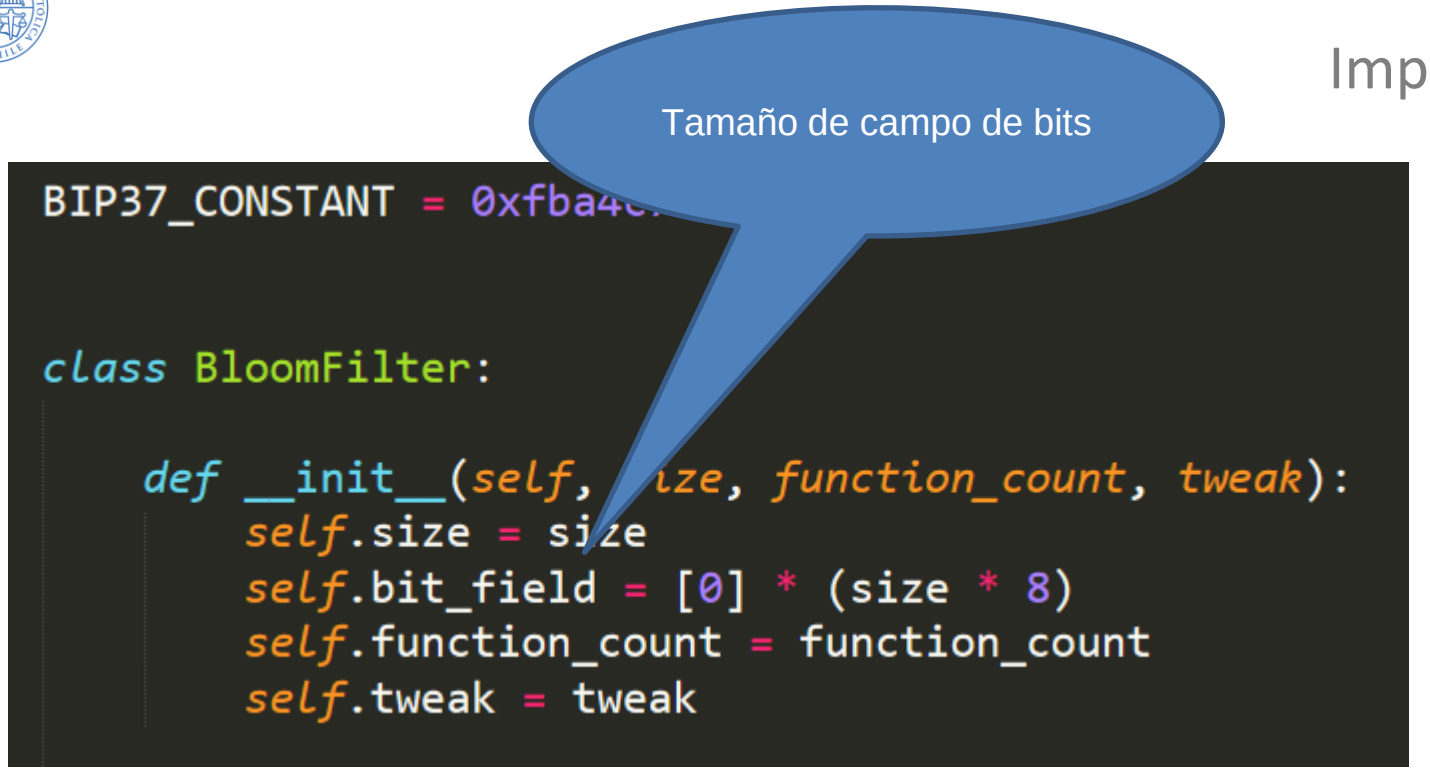
Implementación



bloomfilter.py

Filtros de Bloom

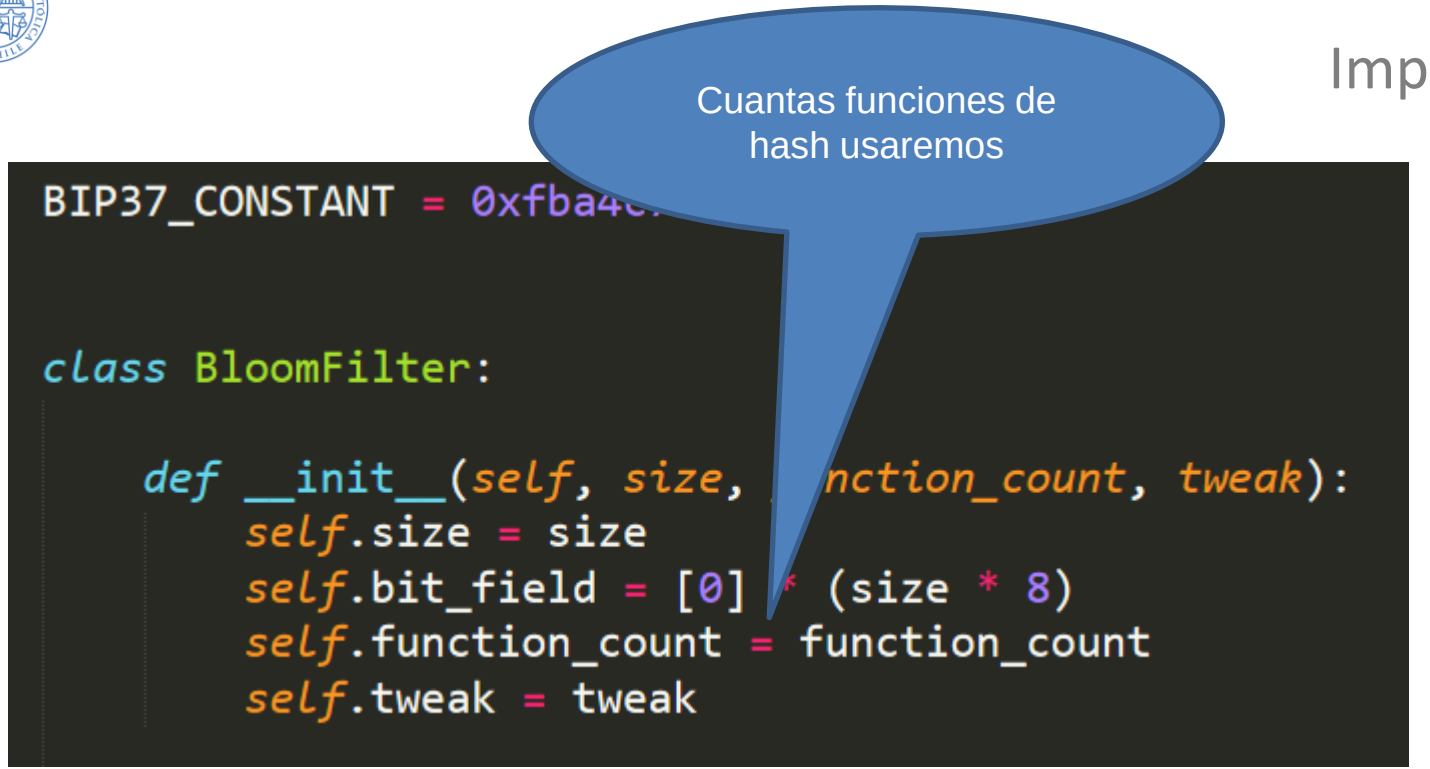
Implementación



bloomfilter.py

Filtros de Bloom

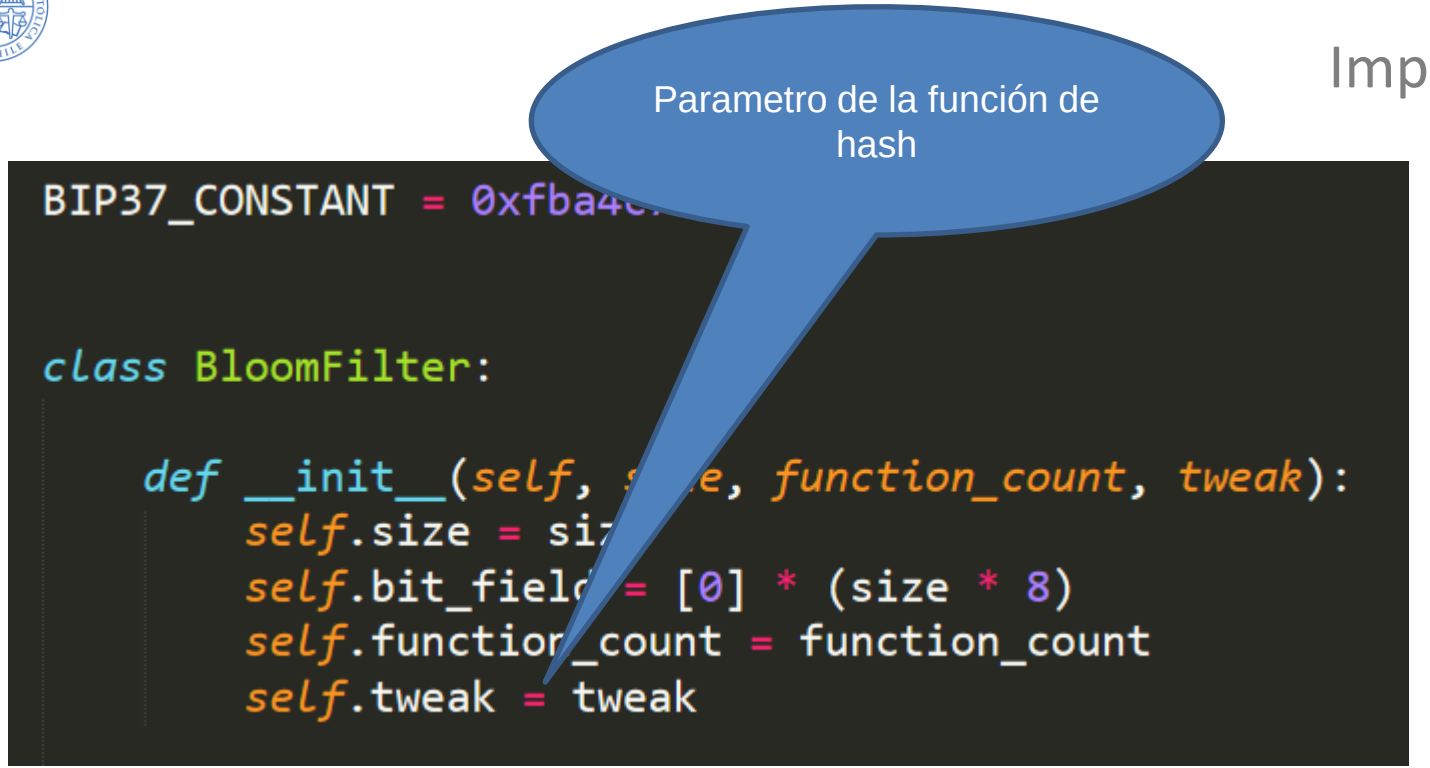
Implementación



bloomfilter.py

Filtros de Bloom

Implementación



bloomfilter.py

Filtros de Bloom

Implementación

Función de hash para Filtros de Bloom:

- Siempre se usa ***murmur3***
- Una función que no es criptográficamente segura
- Pero si distribuye bien a los datos y es super rápida de computar

murmur3 toma un seed:

- i * 0xfba4c795 + tweak

Filtros de Bloom

Implementación

Función de hash para Filtros de Bloom:

- Siempre se usa ***murmur3***
- Una función que no es criptográficamente segura
- Pero si distribuye bien a los datos y es super rápida de computar

murmur3 toma un seed:

- i * 0xfba4c795 + tweak

i = 0 primera función
i = 1 segunda función
….

Filtros de Bloom

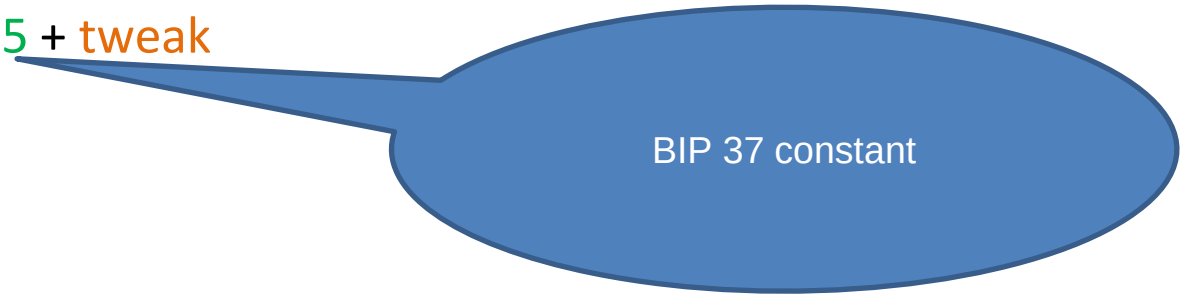
Implementación

Función de hash para Filtros de Bloom:

- Siempre se usa ***murmur3***
- Una función que no es criptográficamente segura
- Pero si distribuye bien a los datos y es super rápida de computar

murmur3 toma un seed:

- i * 0xfba4c795 + tweak



Filtros de Bloom

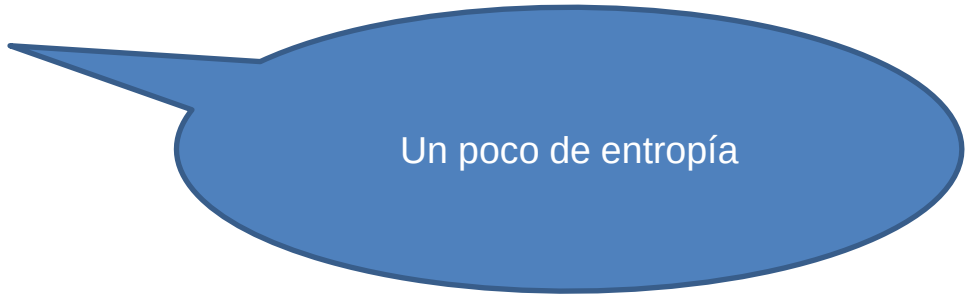
Implementación

Función de hash para Filtros de Bloom:

- Siempre se usa ***murmur3***
- Una función que no es criptográficamente segura
- Pero si distribuye bien a los datos y es super rápida de computar

murmur3 toma un seed:

- i * 0xfba4c795 + tweak



Filtros de Bloom

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

    ...
```

Implementada en helper.py

Filtros de Bloom

Agregar un dato

```
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
```

Un mensaje en la red de Bitcoin

Forma de todos los mensajes

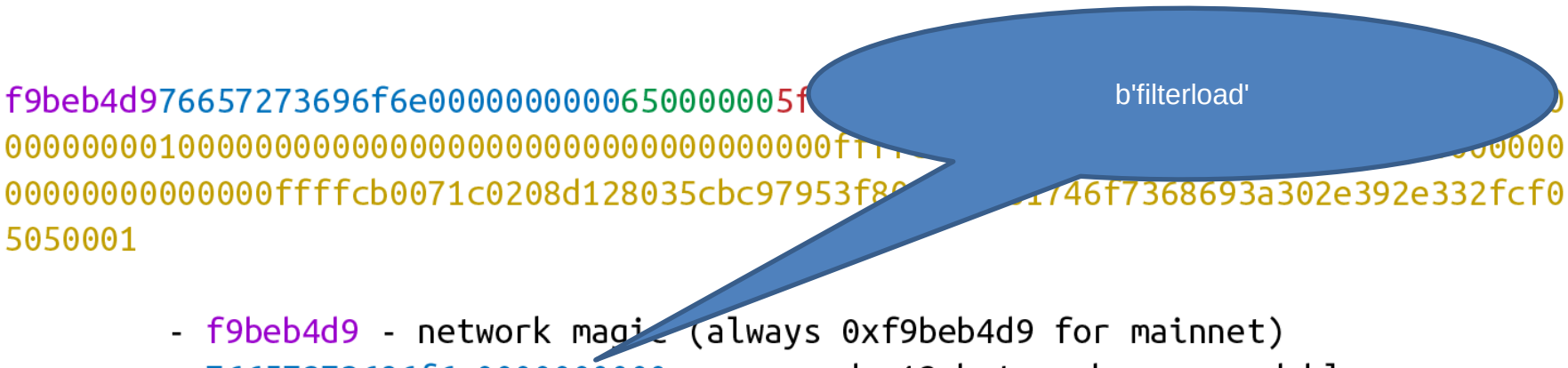
```
f9beb4d976657273696f6e0000000000650000005f1a69d2721101000100000000000000bc8f5e540
0000000010000000000000000000000000000000000ffffc61b6409208d010000000000000000000000
00000000000000ffffcb0071c0208d128035cbc97953f80f2f5361746f7368693a302e392e332fcf0
5050001
```

- 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

Un mensaje en la red de Bitcoin

Forma de todos los mensajes

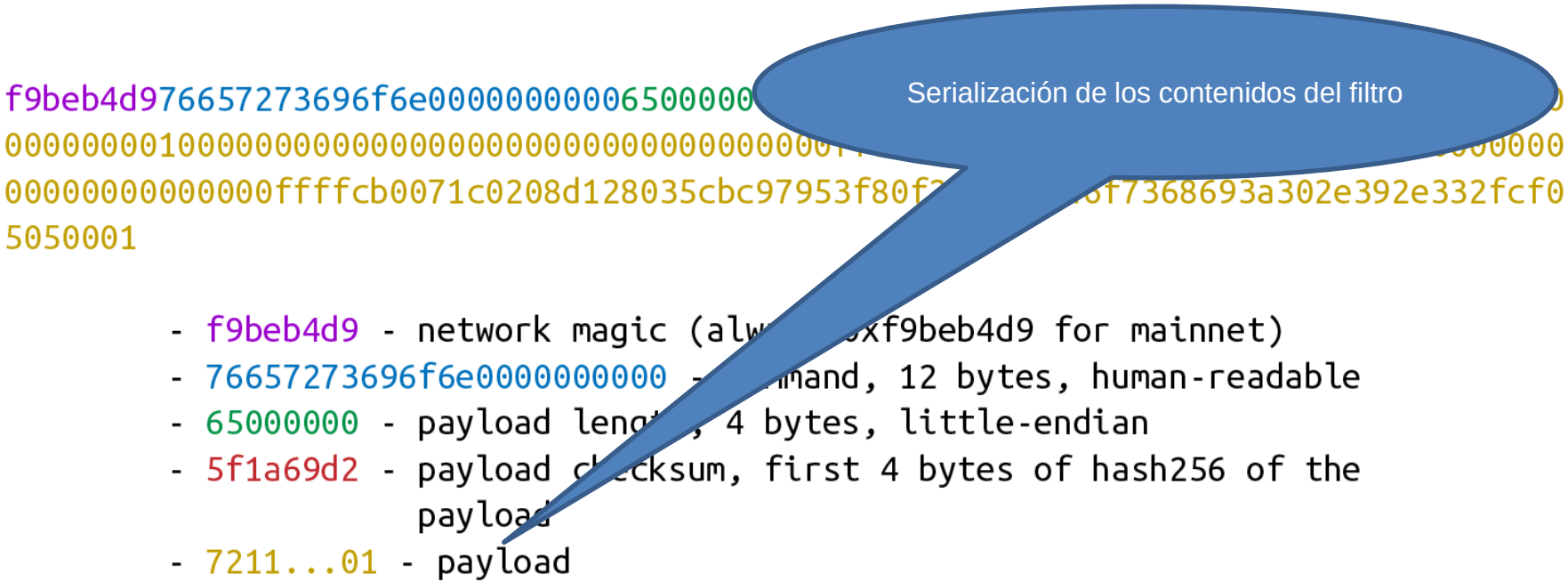


- f9beb4d9 - network magic (always 0xf9beb4d9 for mainnet)
- 76657273696f6e0000000000 - command, 12 bytes, human-readable
- 65000000 - payload length, 4 bytes, little-endian
- 5f1a69d2 - payload checksum, first 4 bytes of hash256 of the payload
- 7211...01 - payload

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

Un mensaje en la red de Bitcoin

Forma de todos los mensajes



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

Filtros de Bloom

Payload de 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

Filtros de Bloom

Payload de filterload

Bitfield del filtro
Formato: varint(largo) + bits_in_bytes

0a4000600a080000010940[illegible]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

Filtros de Bloom

Payload de filterload

Numero de funciones de hash usadas para este filtro

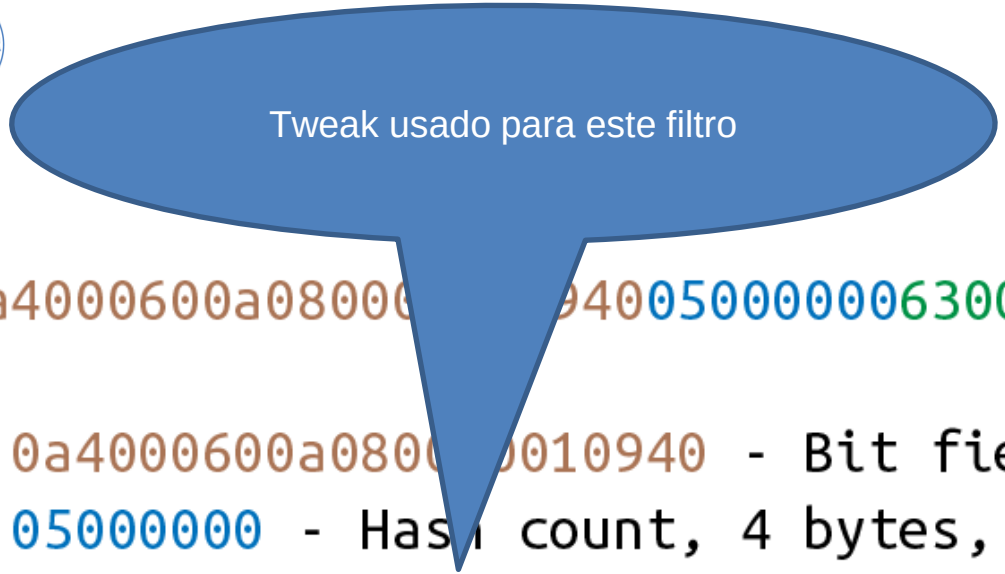
0a4000600a080000[illegible]0050000006300000000

- 0a4000600a080000[illegible]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

Filtros de Bloom

Payload de filterload



0a4000600a08000[illegible]940050000006300000000

- 0a4000600a0800[illegible]010940 - 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

Filtros de Bloom

Payload de filterload

Flag que especifica cómo uno agrega un elemento al filtro

```
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

Filtros de Bloom

Agregar un dato

```
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)
```

Filtros de Bloom

Agregar un dato

```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

Filtros de Bloom

En una conexión

```
# 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

Pedir pruebas de Merkle

getdata

Un mensaje getdata con el payload:

```
02030000030eb2540c41025690160a1014c577061596e32e426b712c7ca000
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

Pedir pruebas de Merkle

getdata

Cuántos?
Formato: varint

Un mensaje getdata con [illegible]

```
02030000003 0eb2540c41025690160a1014c577061596e32e426b712c7ca000
0000000000003000000 1049847939585b0652fba793661c361223446b6fc410
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

Pedir pruebas de Merkle

getdata

Tipo de datos
3 = MerkleProof
(para las cosas filtradas por BF)

Un mensaje getdata con el

```
02030000030eb2540c41025690160a1014c577061596e32e426b712c7ca000
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

Pedir pruebas de Merkle

getdata

Hash de la cosa que se pide

Un mensaje getdata con el paylo

```
02030000030eb2540c41025690160a1014c577061596e32e426b712c7ca000
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

Pedir pruebas de Merkle

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

Pedir pruebas de Merkle

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
```

¿Qué recibo como la respuesta?

merkleblock

spv.py

Preba de Merkle

merkleblock

```
00000020df3b053dc46f162a9b00c7f0d5124e2676d47bbe7c5d0793a500000000000000ef445fef2
ed495c275892206ca533e7411907971013ab83e3b47bd0d692d14d4dc7c835b67d8001ac157e670bf
0d00000aba412a0d1480e370173072c9562becffe87aa661c1e4a6dbc305d38ec5dc088a7cf92e645
8aca7b32edae818f9c2c98c37e06bf72ae0ce80649a38655ee1e27d34d9421d940b16732f24b94023
e9d572a7f9ab8023434a4feb532d2adfc8c2c2158785d1bd04eb99df2e86c54bc13e1398628972174
00def5d72c280222c4cbaee7261831e1550dbb8fa82853e9fe506fc5fda3f7b919d8fe74b6282f927
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

Preba de Merkle

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

Preba de Merkle

merkleblock

Es un mensaje
(como todas las cosas en 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

Preba de Merkle

merkleblock

```
class Mer
    comma

    def _
```

```
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)
```

```
s, flags):
```

merkleblock.py

Preba de Merkle

merkleblock

Para validar la prueba

```python
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

Preba de Merkle

merkleblock

Para validar la prueba

```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
```

¿Qué es esto?

Clase 6
(flag bits!!!)

merkleblock.py

Preba de Merkle

merkleblock

```python
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

Preba de Merkle

merkleblock

```
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:
```

Clase 6

…

merkleblock.py

Preba de Merkle

transacciones

Con getadata para MerkleProofs puedo recibir una transacción también:

- ¿Sirve nuestra implementación de Tx como para mandar un mensaje de red?

¡Obvio que sí!

Cada cosa se serializa/parsea como se manda por la red

Solo hay que agregar una palabra a nuestra implementación tx.py

Preba de Merkle

Transacciones - antes

```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

Preba de Merkle

Transacciones – como un mensaje

It is really that easy!

```
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

Nodo SPV

filterload

```
# 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

Nodo SPV

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

Nodo SPV

Respuestas

```
# 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

Nodo SPV

Respuestas

```
# 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 (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))
```

Recibiremos:
-MerkleBlock
-Tx

spv.py

Nodo SPV

Respuestas

```
# 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))
```

En caso de MerklBlock: Hay que validar la prueba

Si nuestra dirección no está en este bloque la prueba tiene solo el MerkleRoot

spv.py

Nodo SPV

Respuestas

```
# 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))
```

En caso de Tx

spv.py

Nodo SPV

Respuestas

```
# 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))
```

Buscamos nuestra dirección (Método address implementado en script.py)

spv.py

Bitcoin

Nuestra implementación

Para correr un nodo:

- **network.py –** para mandar mensajes
- **bloomfilter.py –** para manejar filtros de Bloom en conexiones
- **merkleblock.py –** para manejar pruebas de merkle
- **block.py –** para manejar los headers y los bloques enteros
- **tx.py –** para manejar transacciones
- **script.py –** para procesar inputs/outputs de transacciones
- **op.py –** para procesar comandos de Script
- **ecc.py** – para criptografía
- **helper.py –** para funciones que faltan (coding/decoding)

Referencias

Leer:

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