The Bitcoin Network

Peer-to-Peer Network Architecture

Bitcoin is structured as a peer-to-peer network architecture on top of the internet. The term peer-to-peer, or P2P, means that the computers that participate in the network are peers to each other, that they are all equal, that there are no "special" nodes, and that all nodes share the burden of providing network services. The network nodes interconnect in a mesh network with a "flat" topology. There is no server, no centralized service, and no hierarchy within the network. Nodes in a P2P network both provide and consume services at the same time with reciprocity acting as the incentive for participation. P2P networks are inherently resilient, decentralized, and open. A preeminent example of a P2P network architecture was the early internet itself, where nodes on the IP network were equal. Today's internet architecture is more hierarchical, but the Internet Protocol still retains its flat-topology essence. Beyond bitcoin, the largest and most successful application of P2P technologies is file sharing, with Napster as the pioneer and BitTorrent as the most recent evolution of the architecture.

Bitcoin's P2P network architecture is much more than a topology choice. Bitcoin is a P2P digital cash system by design, and the network architecture is both a reflection and a foundation of that core characteristic. Decentralization of control is a core design principle that can only be achieved and maintained by a flat, decentralized P2P consensus network.

The term "bitcoin network" refers to the collection of nodes running the bitcoin P2P protocol. In addition to the bitcoin P2P protocol, there are other protocols such as Stratum that are used for mining and lightweight or mobile wallets. These additional protocols are provided by gateway routing servers that access the bitcoin network using the bitcoin P2P protocol and then extend that network to nodes running other protocols. For example, Stratum servers connect Stratum mining nodes via the Stratum protocol to the main bitcoin network and bridge the Stratum protocol to the bitcoin P2P protocol. We use the term "extended bitcoin network" to refer to the overall network that includes the bitcoin P2P protocol, pool-mining protocols, the Stratum protocol, and any other related protocols connecting the components of the bitcoin system.

Node Types and Roles

Although nodes in the bitcoin P2P network are equal, they may take on different roles depending on the functionality they are supporting. A bitcoin node is a collection of functions: routing, the blockchain database, mining, and wallet services. A full node with all four of these functions is shown in A bitcoin network node with all four functions: wallet, miner, full blockchain database, and network routing.

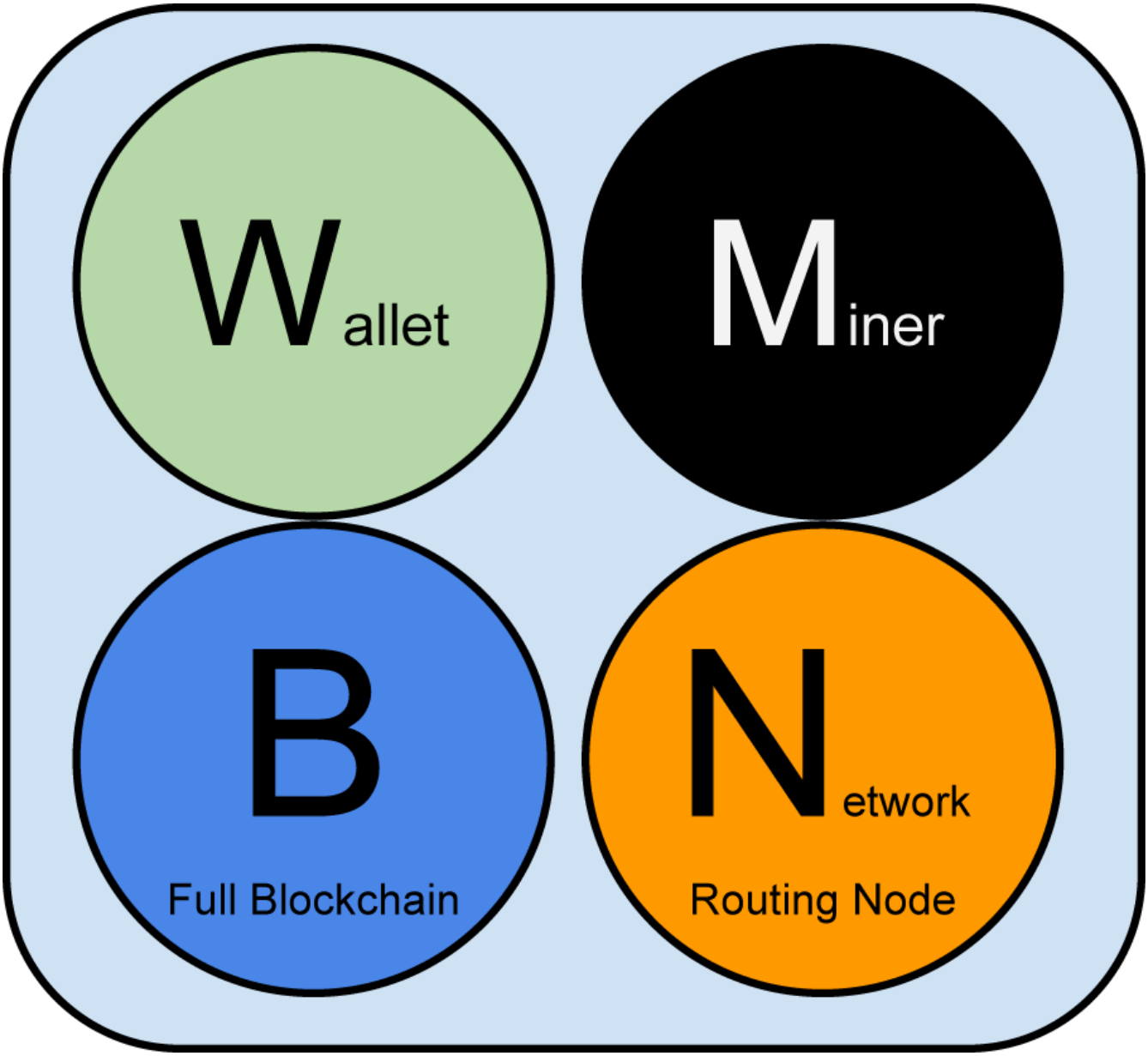


Figure 1. A bitcoin network node with all four functions: wallet, miner, full blockchain database, and network routing

All nodes include the routing function to participate in the network and might include other functionality. All nodes validate and propagate transactions and blocks, and discover and maintain connections to peers. In the full-node example in A bitcoin network node with all four functions: wallet, miner, full blockchain database, and network routing, the routing function is indicated by a circle named "Network Routing Node" or with the letter "N."

Some nodes, called full nodes, also maintain a complete and up-to-date copy of the blockchain. Full nodes can autonomously and authoritatively verify any transaction without external reference. Some nodes maintain only a subset of the blockchain and verify transactions using a method called *simplified payment verification*, or SPV. These nodes are known as SPV nodes or lightweight nodes. In the full-node example in the figure, the full-node blockchain database function is indicated by a circle called "Full Blockchain" or the letter "B." In The extended bitcoin network showing various node types, gateways, and protocols, SPV nodes are drawn without the "B" circle, showing that they do not have a full copy of the blockchain.

Mining nodes compete to create new blocks by running specialized hardware to solve the Proof-of-

Work algorithm. Some mining nodes are also full nodes, maintaining a full copy of the blockchain, while others are lightweight nodes participating in pool mining and depending on a pool server to maintain a full node. The mining function is shown in the full node as a circle called "Miner" or the letter "M."

User wallets might be part of a full node, as is usually the case with desktop bitcoin clients. Increasingly, many user wallets, especially those running on resource-constrained devices such as smartphones, are SPV nodes. The wallet function is shown in A bitcoin network node with all four functions: wallet, miner, full blockchain database, and network routing as a circle called "Wallet" or the letter "W."

In addition to the main node types on the bitcoin P2P protocol, there are servers and nodes running other protocols, such as specialized mining pool protocols and lightweight client-access protocols.

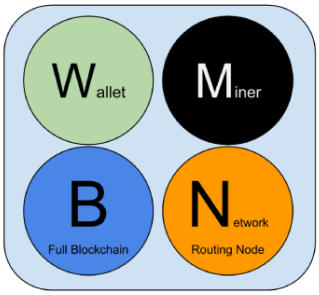
Different types of nodes on the extended bitcoin network shows the most common node types on the extended bitcoin network.

The Extended Bitcoin Network

The main bitcoin network, running the bitcoin P2P protocol, consists of between 5,000 and 8,000 listening nodes running various versions of the bitcoin reference client (Bitcoin Core) and a few hundred nodes running various other implementations of the bitcoin P2P protocol, such as Bitcoin Classic, Bitcoin Unlimited, BitcoinJ, Libbitcoin, btcd, and bcoin. A small percentage of the nodes on the bitcoin P2P network are also mining nodes, competing in the mining process, validating transactions, and creating new blocks. Various large companies interface with the bitcoin network by running full-node clients based on the Bitcoin Core client, with full copies of the blockchain and a network node, but without mining or wallet functions. These nodes act as network edge routers, allowing various other services (exchanges, wallets, block explorers, merchant payment processing) to be built on top.

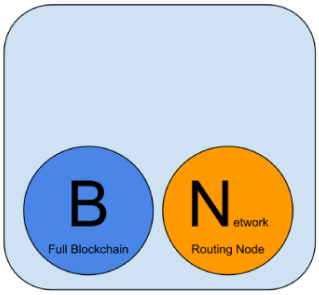
The extended bitcoin network includes the network running the bitcoin P2P protocol, described earlier, as well as nodes running specialized protocols. Attached to the main bitcoin P2P network are a number of pool servers and protocol gateways that connect nodes running other protocols. These other protocol nodes are mostly pool mining nodes (see [mining]) and lightweight wallet clients, which do not carry a full copy of the blockchain.

The extended bitcoin network showing various node types, gateways, and protocols shows the extended bitcoin network with the various types of nodes, gateway servers, edge routers, and wallet clients and the various protocols they use to connect to each other.



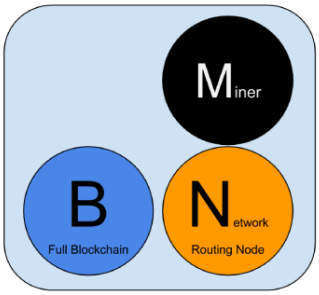
Reference Client (Bitcoin Core)

Contains a Wallet, Miner, full Blockchain database, and Network routing node on the bitcoin P2P network.



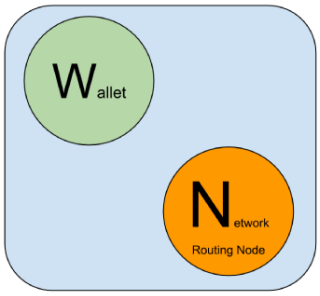
Full Block Chain Node

Contains a full Blockchain database, and Network routing node on the bitcoin P2P network.



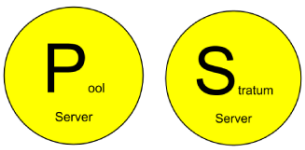
Solo Miner

Contains a mining function with a full copy of the blockchain and a bitcoin P2P network routing node.



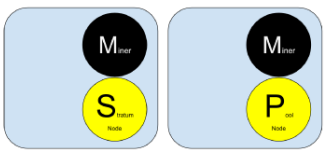
Lightweight (SPV) wallet

Contains a Wallet and a Network node on the bitcoin P2P protocol, without a blockchain.



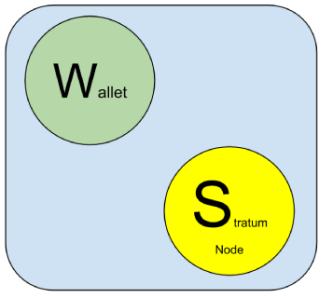
Pool Protocol Servers

Gateway routers connecting the bitcoin P2P network to nodes running other protocols such as pool mining nodes or Stratum nodes.



Mining Nodes

Contain a mining function, without a blockchain, with the Stratum protocol node (S) or other pool (P) mining protocol node.



Lightweight (SPV) Stratum wallet

Contains a Wallet and a Network node on the Stratum protocol, without a blockchain.

Figure 2. Different types of nodes on the extended bitcoin network

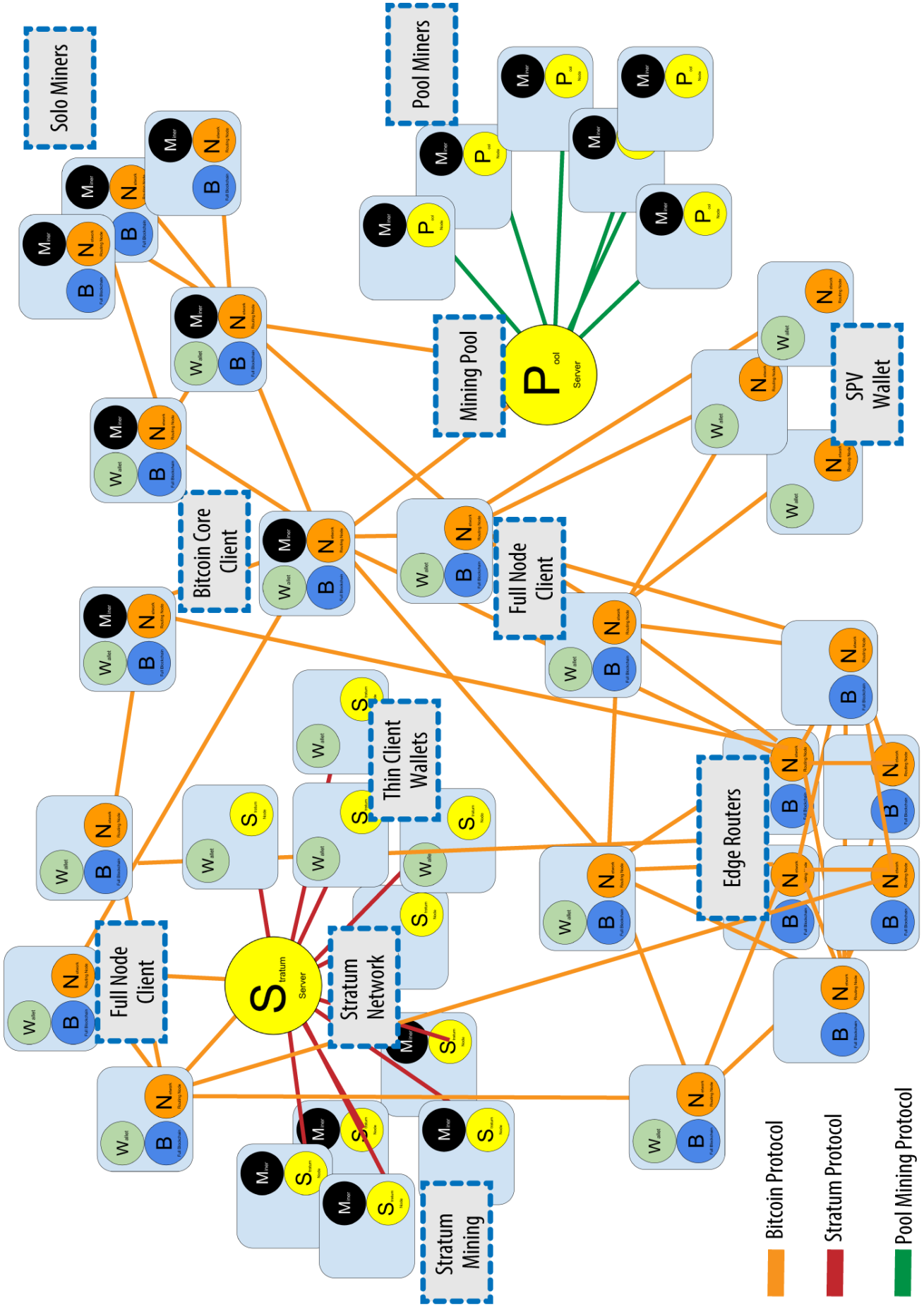


Figure 3. The extended bitcoin network showing various node types, gateways, and protocols

Bitcoin Relay Networks

While the bitcoin P2P network serves the general needs of a broad variety of node types, it exhibits too high network latency for the specialized needs of bitcoin mining nodes.

Bitcoin miners are engaged in a time-sensitive competition to solve the Proof-of-Work problem and extend the blockchain (see [mining]). While participating in this competition, bitcoin miners must minimize the time between the propagation of a winning block and the beginning of the next round of competition. In mining, network latency is directly related to profit margins.

A *Bitcoin Relay Network* is a network that attempts to minimize the latency in the transmission of blocks between miners. The original Bitcoin Relay Network was created by core developer Matt Corallo in 2015 to enable fast synchronization of blocks between miners with very low latency. The network consisted of several specialized nodes hosted on the Amazon Web Services infrastructure around the world and served to connect the majority of miners and mining pools.

The original Bitcoin Relay Network was replaced in 2016 with the introduction of the *Fast Internet Bitcoin Relay Engine* or *FIBRE*, also created by core developer Matt Corallo. FIBRE is a UDP-based relay network that relays blocks within a network of nodes. FIBRE implements *compact block* optimization to further reduce the amount of data transmitted and the network latency.

Another relay network (still in the proposal phase) is *Falcon*, based on research at Cornell University. Falcon uses "cut-through-routing" instead of "store-and-forward" to reduce latency by propagating parts of blocks as they are received rather than waiting until a complete block is received.

Relay networks are not replacements for bitcoin's P2P network. Instead they are overlay networks that provide additional connectivity between nodes with specialized needs. Like freeways are not replacements for rural roads, but rather shortcuts between two points with heavy traffic, you still need small roads to connect to the freeways.

Network Discovery

When a new node boots up, it must discover other bitcoin nodes on the network in order to participate. To start this process, a new node must discover at least one existing node on the network and connect to it. The geographic location of other nodes is irrelevant; the bitcoin network topology is not geographically defined. Therefore, any existing bitcoin nodes can be selected at random.

To connect to a known peer, nodes establish a TCP connection, usually to port 8333 (the port generally known as the one used by bitcoin), or an alternative port if one is provided. Upon establishing a connection, the node will start a "handshake" (see The initial handshake between peers) by transmitting a version message, which contains basic identifying information, including:

nVersion

The bitcoin P2P protocol version the client "speaks" (e.g., 70002)

nLocalServices

A list of local services supported by the node, currently just NODE_NETWORK

nTime

The current time

addrYou

The IP address of the remote node as seen from this node

addrMe

The IP address of the local node, as discovered by the local node

subver

A sub-version showing the type of software running on this node (e.g., `/Satoshi:0.9.2.1/`)

BestHeight

The block height of this node's blockchain

(See GitHub for an example of the version network message.)

The version message is always the first message sent by any peer to another peer. The local peer receiving a version message will examine the remote peer's reported nVersion and decide if the remote peer is compatible. If the remote peer is compatible, the local peer will acknowledge the version message and establish a connection by sending a verack message.

How does a new node find peers? The first method is to query DNS using a number of "DNS seeds," which are DNS servers that provide a list of IP addresses of bitcoin nodes. Some of those DNS seeds provide a static list of IP addresses of stable bitcoin listening nodes. Some of the DNS seeds are custom implementations of BIND (Berkeley Internet Name Daemon) that return a random subset from a list of bitcoin node addresses collected by a crawler or a long-running bitcoin node. The Bitcoin Core client contains the names of nine different DNS seeds. The diversity of ownership and diversity of implementation of the different DNS seeds offers a high level of reliability for the initial bootstrapping process. In the Bitcoin Core client, the option to use the DNS seeds is controlled by the option switch -dnsseed (set to 1 by default, to use the DNS seed).

Alternatively, a bootstrapping node that knows nothing of the network must be given the IP address of at least one bitcoin node, after which it can establish connections through further introductions. The command-line argument -seednode can be used to connect to one node just for introductions using it as a seed. After the initial seed node is used to form introductions, the client will disconnect from it and use the newly discovered peers.

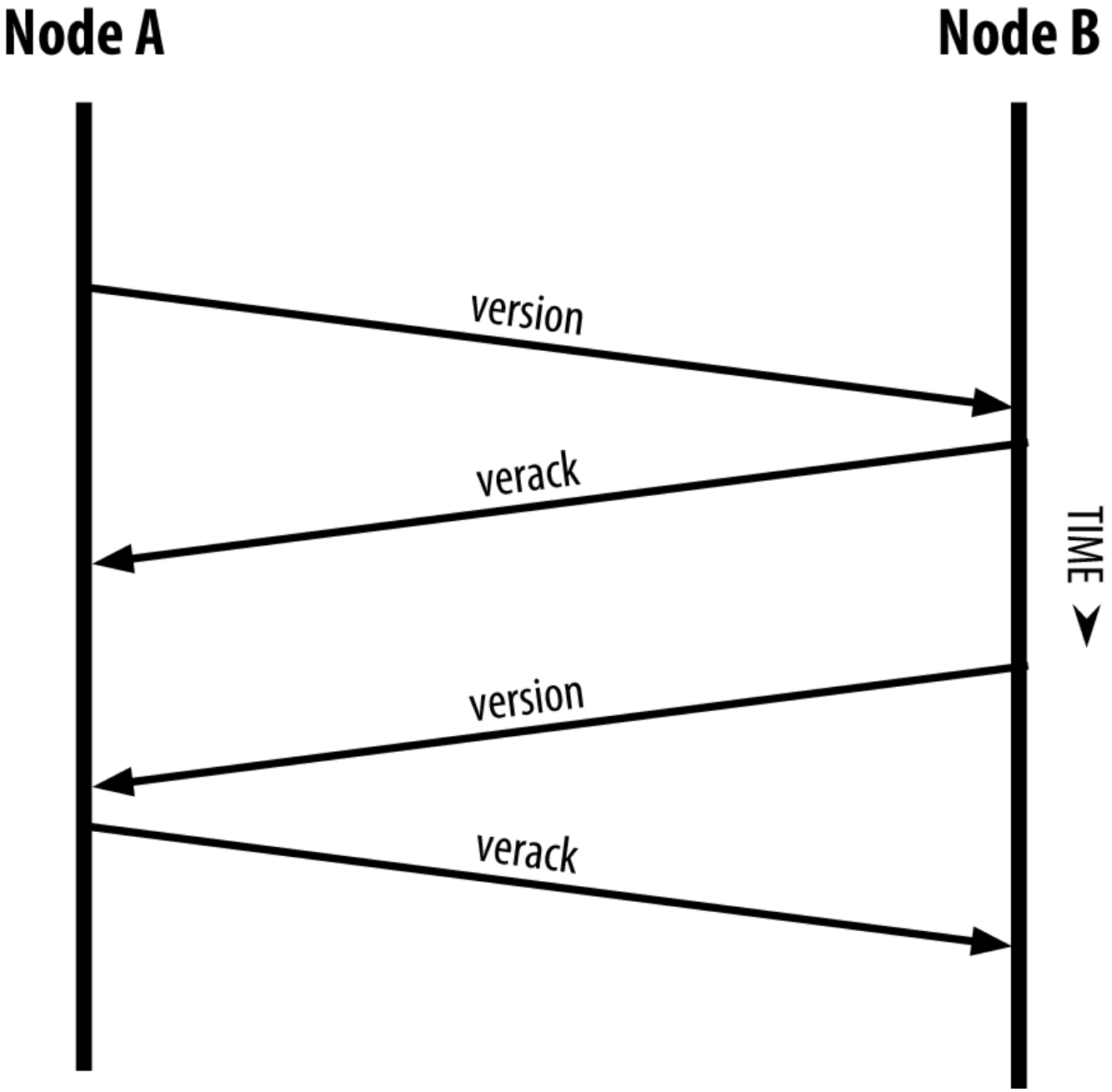


Figure 4. The initial handshake between peers

Once one or more connections are established, the new node will send an addr message containing its own IP address to its neighbors. The neighbors will, in turn, forward the addr message to their neighbors, ensuring that the newly connected node becomes well known and better connected. Additionally, the newly connected node can send getaddr to the neighbors, asking them to return a list of IP addresses of other peers. That way, a node can find peers to connect to and advertise its existence on the network for other nodes to find it. Address propagation and discovery shows the address discovery protocol.

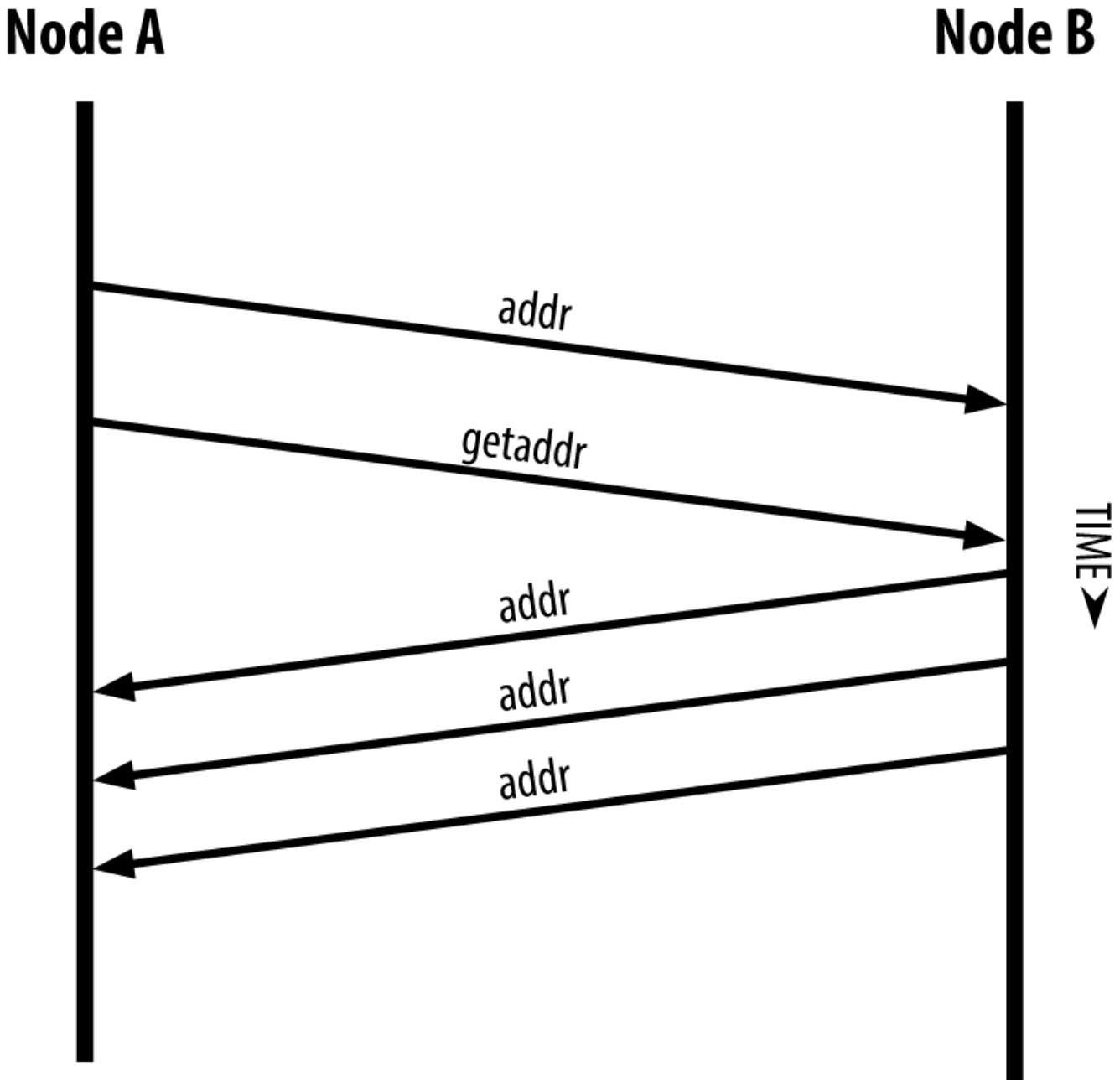


Figure 5. Address propagation and discovery

A node must connect to a few different peers in order to establish diverse paths into the bitcoin network. Paths are not persistent—nodes come and go—and so the node must continue to discover new nodes as it loses old connections as well as assist other nodes when they bootstrap. Only one connection is needed to bootstrap, because the first node can offer introductions to its peer nodes and those peers can offer further introductions. It's also unnecessary and wasteful of network resources to connect to more than a handful of nodes. After bootstrapping, a node will remember its most recent successful peer connections, so that if it is rebooted it can quickly reestablish connections with its former peer network. If none of the former peers respond to its connection request, the node can use the seed nodes to bootstrap again.

On a node running the Bitcoin Core client, you can list the peer connections with the command getpeerinfo:

```
$ bitcoin-cli getpeerinfo
```

```
[
    {
        "addr" : "85.213.199.39:8333",
        "services" : "00000001",
        "lastsend" : 1405634126,
        "lastrecv" : 1405634127,
        "bytessent" : 23487651,
        "bytesrecv" : 138679099,
        "conntime" : 1405021768,
        "pingtime" : 0.00000000,
        "version" : 70002,
        "subver" : "/Satoshi:0.9.2.1/",
        "inbound" : false,
        "startingheight" : 310131,
        "banscore" : 0,
        "syncnode" : true
    },
    {
        "addr" : "58.23.244.20:8333",
        "services" : "00000001",
        "lastsend" : 1405634127,
        "lastrecv" : 1405634124,
        "bytessent" : 4460918,
        "bytesrecv" : 8903575,
        "conntime" : 1405559628,
        "pingtime" : 0.00000000,
        "version" : 70001,
        "subver" : "/Satoshi:0.8.6/",
        "inbound" : false,
        "startingheight" : 311074,
        "banscore" : 0,
        "syncnode" : false
    }
]
```


To override the automatic management of peers and to specify a list of IP addresses, users can provide the option -connect=<IPAddress> and specify one or more IP addresses. If this option is used, the node will only connect to the selected IP addresses, instead of discovering and maintaining the peer connections automatically.

If there is no traffic on a connection, nodes will periodically send a message to maintain the connection. If a node has not communicated on a connection for more than 90 minutes, it is assumed to be disconnected and a new peer will be sought. Thus, the network dynamically adjusts to transient nodes and network problems, and can organically grow and shrink as needed without any central control.

Full Nodes

Full nodes are nodes that maintain a full blockchain with all transactions. More accurately, they probably should be called "full blockchain nodes." In the early years of bitcoin, all nodes were full nodes and currently the Bitcoin Core client is a full blockchain node. In the past two years, however, new forms of bitcoin clients have been introduced that do not maintain a full blockchain but run as lightweight clients. We'll examine these in more detail in the next section.

Full blockchain nodes maintain a complete and up-to-date copy of the bitcoin blockchain with all the transactions, which they independently build and verify, starting with the very first block (genesis block) and building up to the latest known block in the network. A full blockchain node can independently and authoritatively verify any transaction without recourse or reliance on any other node or source of information. The full blockchain node relies on the network to receive updates about new blocks of transactions, which it then verifies and incorporates into its local copy of the blockchain.

Running a full blockchain node gives you the pure bitcoin experience: independent verification of all transactions without the need to rely on, or trust, any other systems. It's easy to tell if you're running a full node because it requires more than one hundred gigabytes of persistent storage (disk space) to store the full blockchain. If you need a lot of disk and it takes two to three days to sync to the network, you are running a full node. That is the price of complete independence and freedom from central authority.

There are a few alternative implementations of full blockchain bitcoin clients, built using different programming languages and software architectures. However, the most common implementation is the reference client Bitcoin Core, also known as the Satoshi client. More than 75% of the nodes on the bitcoin network run various versions of Bitcoin Core. It is identified as "Satoshi" in the sub-version string sent in the version message and shown by the command getpeerinfo as we saw earlier; for example, /Satoshi:0.8.6/.

Exchanging "Inventory"

The first thing a full node will do once it connects to peers is try to construct a complete blockchain. If it is a brand-new node and has no blockchain at all, it only knows one block, the genesis block, which is statically embedded in the client software. Starting with block #0 (the genesis block), the new node will have to download hundreds of thousands of blocks to synchronize with the network and reestablish the full blockchain.

The process of syncing the blockchain starts with the version message, because that contains BestHeight, a node's current blockchain height (number of blocks). A node will see the version messages from its peers, know how many blocks they each have, and be able to compare to how many blocks it has in its own blockchain. Peered nodes will exchange a getblocks message that contains the hash (fingerprint) of the top block on their local blockchain. One of the peers will be able to identify the received hash as belonging to a block that is not at the top, but rather belongs to an older block, thus deducing that its own local blockchain is longer than its peer's.

The peer that has the longer blockchain has more blocks than the other node and can identify which blocks the other node needs in order to "catch up." It will identify the first 500 blocks to

share and transmit their hashes using an inv (inventory) message. The node missing these blocks will then retrieve them, by issuing a series of getdata messages requesting the full block data and identifying the requested blocks using the hashes from the inv message.

Let's assume, for example, that a node only has the genesis block. It will then receive an inv message from its peers containing the hashes of the next 500 blocks in the chain. It will start requesting blocks from all of its connected peers, spreading the load and ensuring that it doesn't overwhelm any peer with requests. The node keeps track of how many blocks are "in transit" per peer connection, meaning blocks that it has requested but not received, checking that it does not exceed a limit (MAX_BLOCKS_IN_TRANSIT_PER_PEER). This way, if it needs a lot of blocks, it will only request new ones as previous requests are fulfilled, allowing the peers to control the pace of updates and not overwhelm the network. As each block is received, it is added to the blockchain, as we will see in [blockchain]. As the local blockchain is gradually built up, more blocks are requested and received, and the process continues until the node catches up to the rest of the network.

This process of comparing the local blockchain with the peers and retrieving any missing blocks happens any time a node goes offline for any period of time. Whether a node has been offline for a few minutes and is missing a few blocks, or a month and is missing a few thousand blocks, it starts by sending getblocks, gets an inv response, and starts downloading the missing blocks. Node synchronizing the blockchain by retrieving blocks from a peer shows the inventory and block propagation protocol.

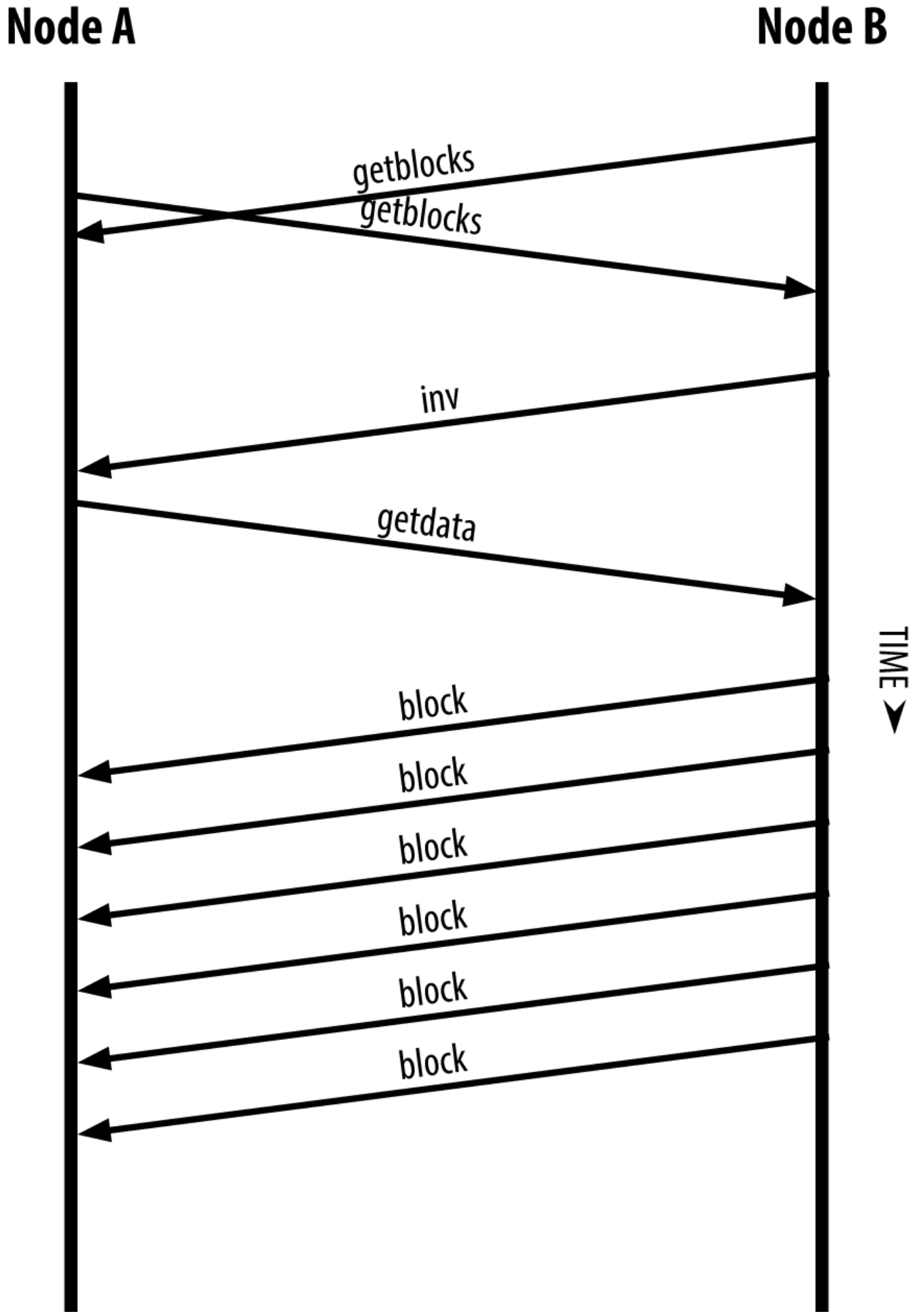


Figure 6. Node synchronizing the blockchain by retrieving blocks from a peer

Simplified Payment Verification (SPV) Nodes

Not all nodes have the ability to store the full blockchain. Many bitcoin clients are designed to run on space- and power-constrained devices, such as smartphones, tablets, or embedded systems. For such devices, a *simplified payment verification* (SPV) method is used to allow them to operate without storing the full blockchain. These types of clients are called SPV clients or lightweight clients. As bitcoin adoption surges, the SPV node is becoming the most common form of bitcoin node, especially for bitcoin wallets.

SPV nodes download only the block headers and do not download the transactions included in each block. The resulting chain of blocks, without transactions, is 1,000 times smaller than the full blockchain. SPV nodes cannot construct a full picture of all the UTXOs that are available for spending because they do not know about all the transactions on the network. SPV nodes verify transactions using a slightly different method that relies on peers to provide partial views of relevant parts of the blockchain on demand.

As an analogy, a full node is like a tourist in a strange city, equipped with a detailed map of every street and every address. By comparison, an SPV node is like a tourist in a strange city asking random strangers for turn-by-turn directions while knowing only one main avenue. Although both tourists can verify the existence of a street by visiting it, the tourist without a map doesn't know what lies down any of the side streets and doesn't know what other streets exist. Positioned in front of 23 Church Street, the tourist without a map cannot know if there are a dozen other "23 Church Street" addresses in the city and whether this is the right one. The mapless tourist's best chance is to ask enough people and hope some of them are not trying to mug him.

SPV verifies transactions by reference to their *depth* in the blockchain instead of their *height*. Whereas a full blockchain node will construct a fully verified chain of thousands of blocks and transactions reaching down the blockchain (back in time) all the way to the genesis block, an SPV node will verify the chain of all blocks (but not all transactions) and link that chain to the transaction of interest.

For example, when examining a transaction in block 300,000, a full node links all 300,000 blocks down to the genesis block and builds a full database of UTXO, establishing the validity of the transaction by confirming that the UTXO remains unspent. An SPV node cannot validate whether the UTXO is unspent. Instead, the SPV node will establish a link between the transaction and the block that contains it, using a *merkle path* (see [merkle_trees]). Then, the SPV node waits until it sees the six blocks 300,001 through 300,006 piled on top of the block containing the transaction and verifies it by establishing its depth under blocks 300,006 to 300,001. The fact that other nodes on the network accepted block 300,000 and then did the necessary work to produce six more blocks on top of it is proof, by proxy, that the transaction was not a double-spend.

An SPV node cannot be persuaded that a transaction exists in a block when the transaction does not in fact exist. The SPV node establishes the existence of a transaction in a block by requesting a merkle path proof and by validating the Proof-of-Work in the chain of blocks. However, a transaction's existence can be "hidden" from an SPV node. An SPV node can definitely prove that a transaction exists but cannot verify that a transaction, such as a double-spend of the same UTXO, doesn't exist because it doesn't have a record of all transactions. This vulnerability can be used in a denial-of-service attack or for a double-spending attack against SPV nodes. To defend against this, an SPV node needs to connect randomly to several nodes, to increase the probability that it is in

contact with at least one honest node. This need to randomly connect means that SPV nodes also are vulnerable to network partitioning attacks or Sybil attacks, where they are connected to fake nodes or fake networks and do not have access to honest nodes or the real bitcoin network.

For most practical purposes, well-connected SPV nodes are secure enough, striking a balance between resource needs, practicality, and security. For infallible security, however, nothing beats running a full blockchain node.

TIP A full blockchain node verifies a transaction by checking the entire chain of thousands of blocks below it in order to guarantee that the UTXO is not spent, whereas an SPV node checks how deep the block is buried by a handful of blocks above it.

To get the block headers, SPV nodes use a getheaders message instead of getblocks. The responding peer will send up to 2,000 block headers using a single headers message. The process is otherwise the same as that used by a full node to retrieve full blocks. SPV nodes also set a filter on the connection to peers, to filter the stream of future blocks and transactions sent by the peers. Any transactions of interest are retrieved using a getdata request. The peer generates a tx message containing the transactions, in response. SPV node synchronizing the block headers shows the synchronization of block headers.

Because SPV nodes need to retrieve specific transactions in order to selectively verify them, they also create a privacy risk. Unlike full blockchain nodes, which collect all transactions within each block, the SPV node's requests for specific data can inadvertently reveal the addresses in their wallet. For example, a third party monitoring a network could keep track of all the transactions requested by a wallet on an SPV node and use those to associate bitcoin addresses with the user of that wallet, destroying the user's privacy.

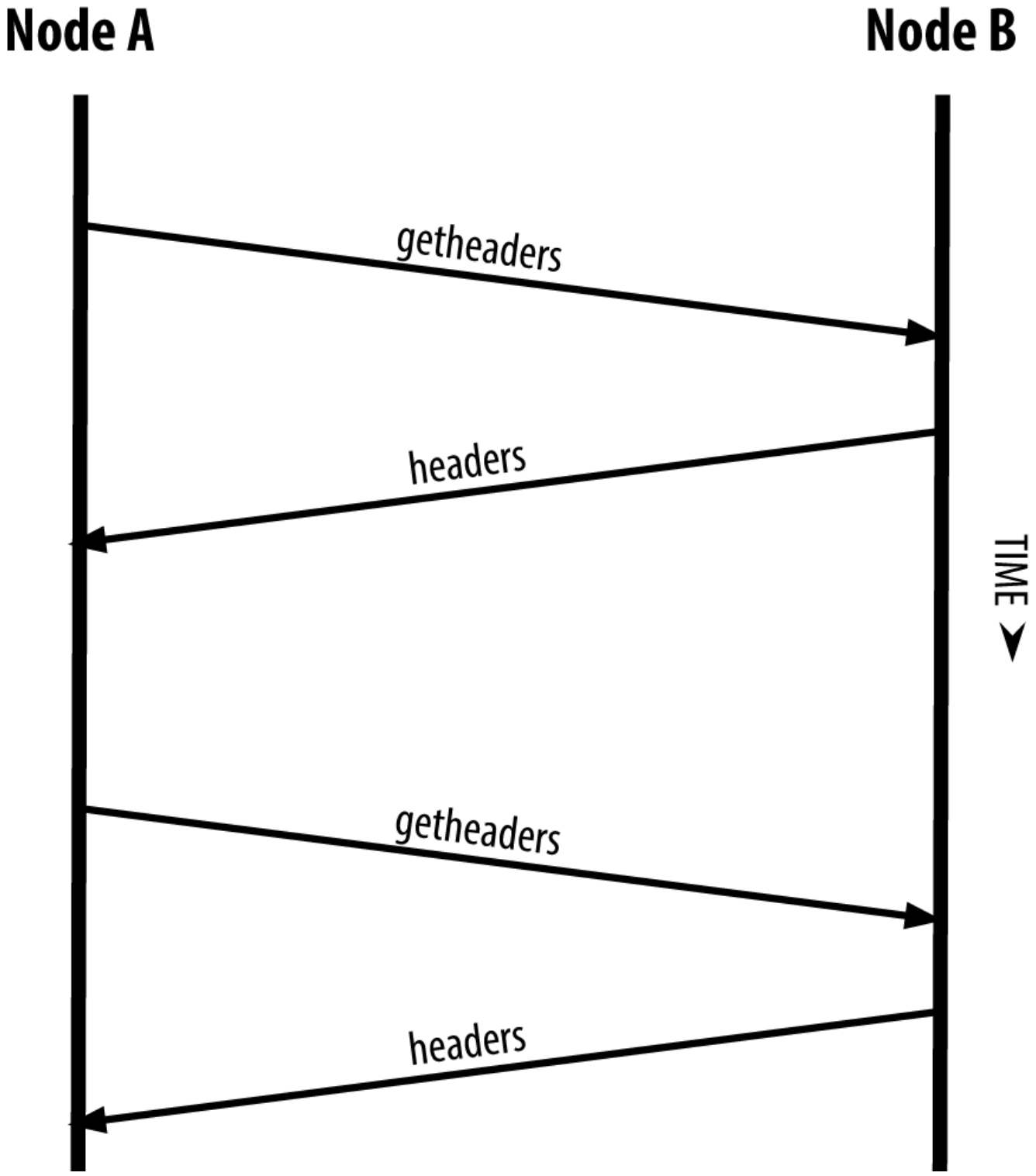


Figure 7. SPV node synchronizing the block headers

Shortly after the introduction of SPV/lightweight nodes, bitcoin developers added a feature called *bloom filters* to address the privacy risks of SPV nodes. Bloom filters allow SPV nodes to receive a subset of the transactions without revealing precisely which addresses they are interested in, through a filtering mechanism that uses probabilities rather than fixed patterns.

Bloom Filters

A bloom filter is a probabilistic search filter that offers an efficient way to express a search pattern while protecting privacy. They are used by SPV nodes to ask their peers for transactions matching a

specific pattern, without revealing exactly which addresses, keys, or transactions they are searching for.

In our previous analogy, a tourist without a map is asking for directions to a specific address, "23 Church St." If she asks strangers for directions to this street, she inadvertently reveals her destination. A bloom filter is like asking, "Are there any streets in this neighborhood whose name ends in R-C-H?" A question like that reveals slightly less about the desired destination than asking for "23 Church St." Using this technique, a tourist could specify the desired address in more detail such as "ending in U-R-C-H" or less detail as "ending in H." By varying the precision of the search, the tourist reveals more or less information, at the expense of getting more or less specific results. If she asks a less specific pattern, she gets a lot more possible addresses and better privacy, but many of the results are irrelevant. If she asks for a very specific pattern, she gets fewer results but loses privacy.

Bloom filters serve this function by allowing an SPV node to specify a search pattern for transactions that can be tuned toward precision or privacy. A more specific bloom filter will produce accurate results, but at the expense of revealing what patterns the SPV node is interested in, thus revealing the addresses owned by the user's wallet. A less specific bloom filter will produce more data about more transactions, many irrelevant to the node, but will allow the node to maintain better privacy.

How Bloom Filters Work

Bloom filters are implemented as a variable-size array of N binary digits (a bit field) and a variable number of M hash functions. The hash functions are designed to always produce an output that is between 1 and N, corresponding to the array of binary digits. The hash functions are generated deterministically, so that any node implementing a bloom filter will always use the same hash functions and get the same results for a specific input. By choosing different length (N) bloom filters and a different number (M) of hash functions, the bloom filter can be tuned, varying the level of accuracy and therefore privacy.

In An example of a simplistic bloom filter, with a 16-bit field and three hash functions, we use a very small array of 16 bits and a set of three hash functions to demonstrate how bloom filters work.

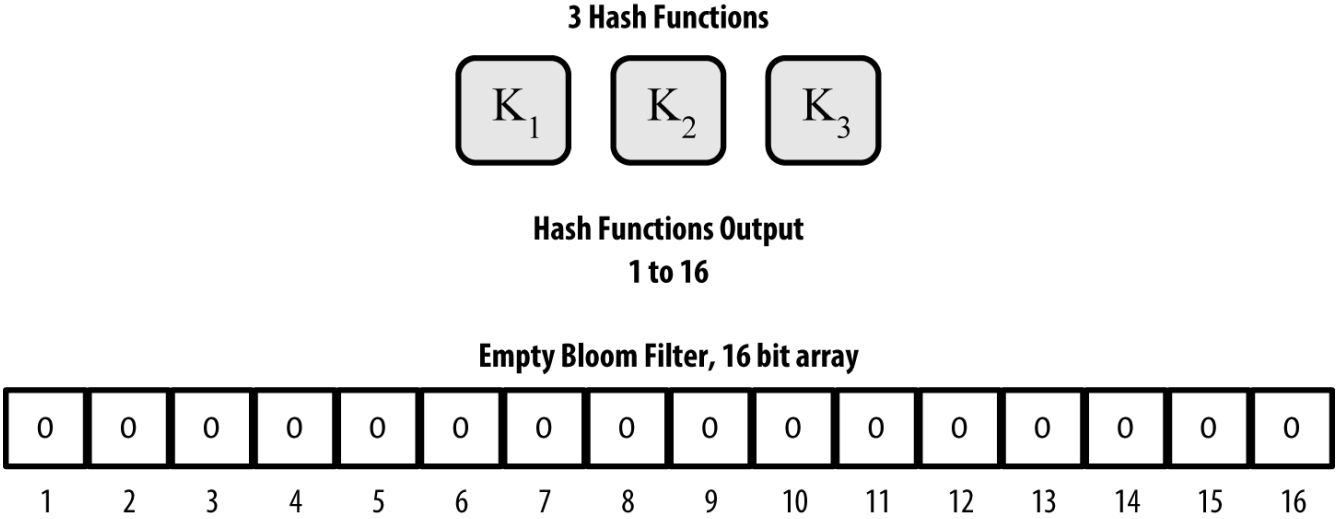


Figure 8. An example of a simplistic bloom filter, with a 16-bit field and three hash functions

The bloom filter is initialized so that the array of bits is all zeros. To add a pattern to the bloom filter, the pattern is hashed by each hash function in turn. Applying the first hash function to the input results in a number between 1 and N. The corresponding bit in the array (indexed from 1 to N) is found and set to 1, thereby recording the output of the hash function. Then, the next hash function is used to set another bit and so on. Once all M hash functions have been applied, the search pattern will be "recorded" in the bloom filter as M bits that have been changed from 0 to 1.

Adding a pattern "A" to our simple bloom filter is an example of adding a pattern "A" to the simple bloom filter shown in An example of a simplistic bloom filter, with a 16-bit field and three hash functions.

Adding a second pattern is as simple as repeating this process. The pattern is hashed by each hash function in turn and the result is recorded by setting the bits to 1. Note that as a bloom filter is filled with more patterns, a hash function result might coincide with a bit that is already set to 1, in which case the bit is not changed. In essence, as more patterns record on overlapping bits, the bloom filter starts to become saturated with more bits set to 1 and the accuracy of the filter decreases. This is why the filter is a probabilistic data structure—it gets less accurate as more patterns are added. The accuracy depends on the number of patterns added versus the size of the bit array (N) and number of hash functions (M). A larger bit array and more hash functions can record more patterns with higher accuracy. A smaller bit array or fewer hash functions will record fewer patterns and produce less accuracy.

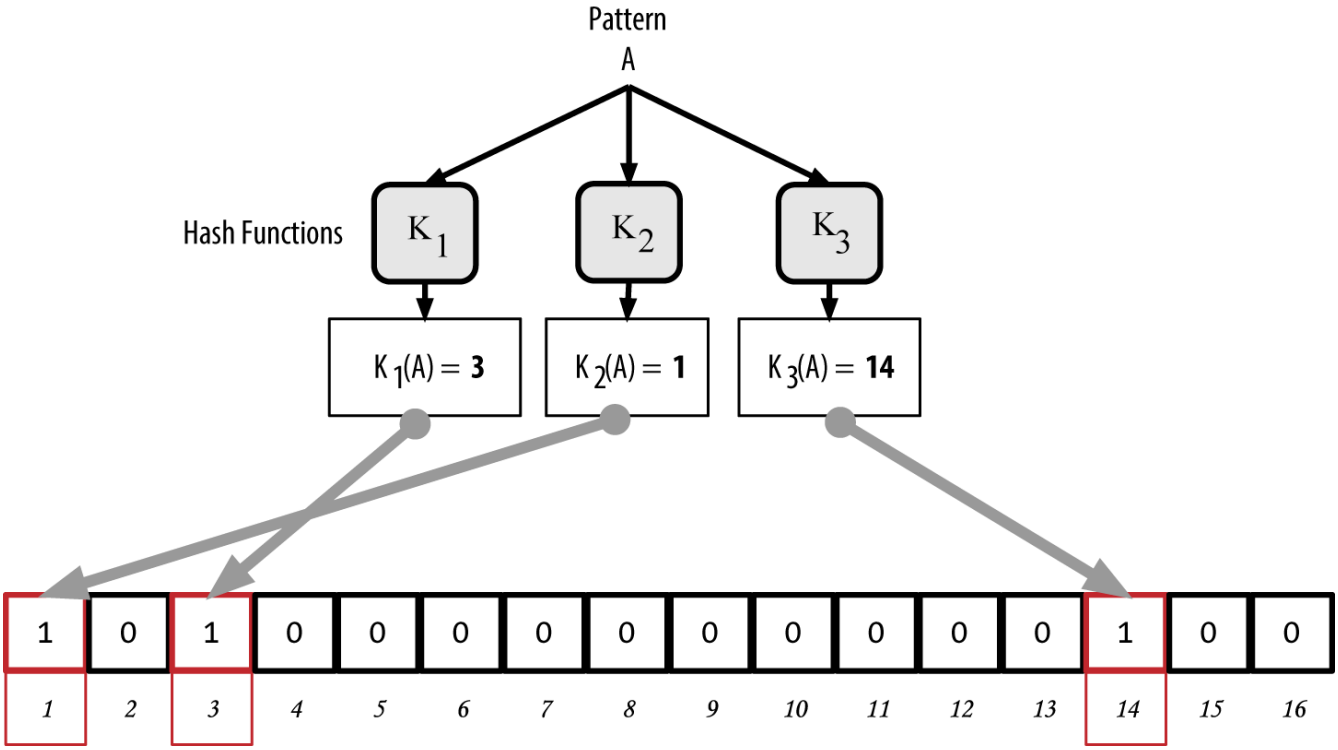


Figure 9. Adding a pattern "A" to our simple bloom filter

Adding a second pattern "B" to our simple bloom filter is an example of adding a second pattern "B" to the simple bloom filter.

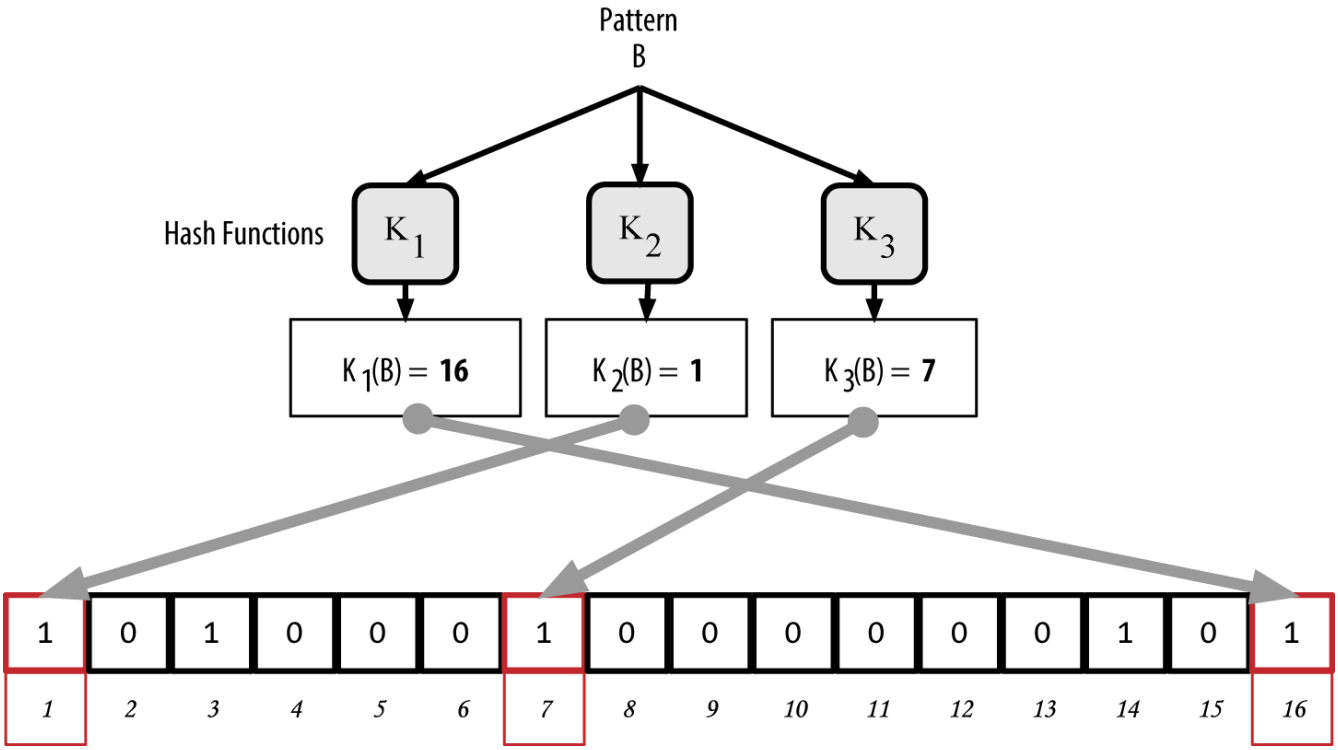


Figure 10. Adding a second pattern "B" to our simple bloom filter

To test if a pattern is part of a bloom filter, the pattern is hashed by each hash function and the resulting bit pattern is tested against the bit array. If all the bits indexed by the hash functions are set to 1, then the pattern is *probably* recorded in the bloom filter. Because the bits may be set because of overlap from multiple patterns, the answer is not certain, but is rather probabilistic. In simple terms, a bloom filter positive match is a "Maybe, Yes."

Testing the existence of pattern "X" in the bloom filter. The result is a probabilistic positive match, meaning "Maybe." is an example of testing the existence of pattern "X" in the simple bloom filter. The corresponding bits are set to 1, so the pattern is probably a match.

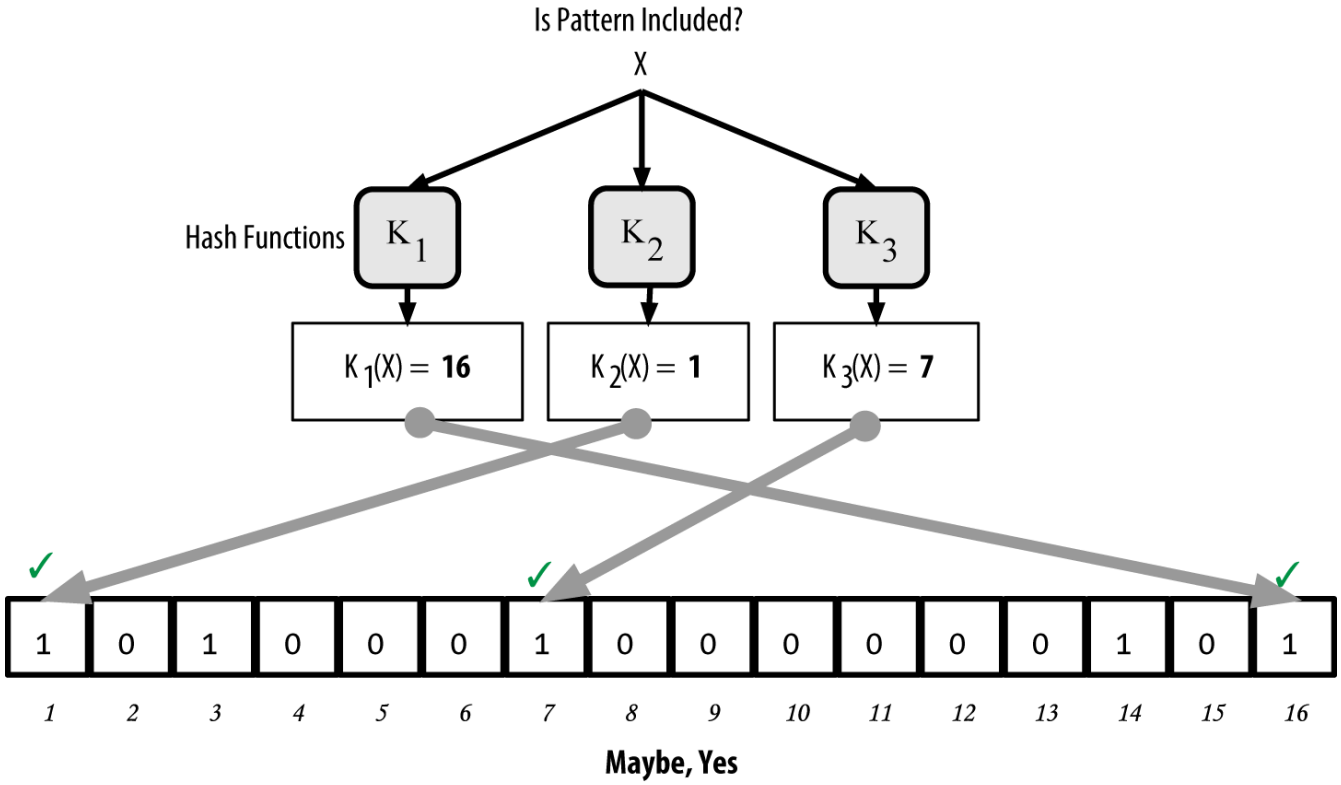


Figure 11. Testing the existence of pattern "X" in the bloom filter. The result is a probabilistic positive match, meaning "Maybe."

On the contrary, if a pattern is tested against the bloom filter and any one of the bits is set to 0, this proves that the pattern was not recorded in the bloom filter. A negative result is not a probability, it is a certainty. In simple terms, a negative match on a bloom filter is a "Definitely Not!"

Testing the existence of pattern "Y" in the bloom filter. The result is a definitive negative match, meaning "Definitely Not!" is an example of testing the existence of pattern "Y" in the simple bloom filter. One of the corresponding bits is set to 0, so the pattern is definitely not a match.

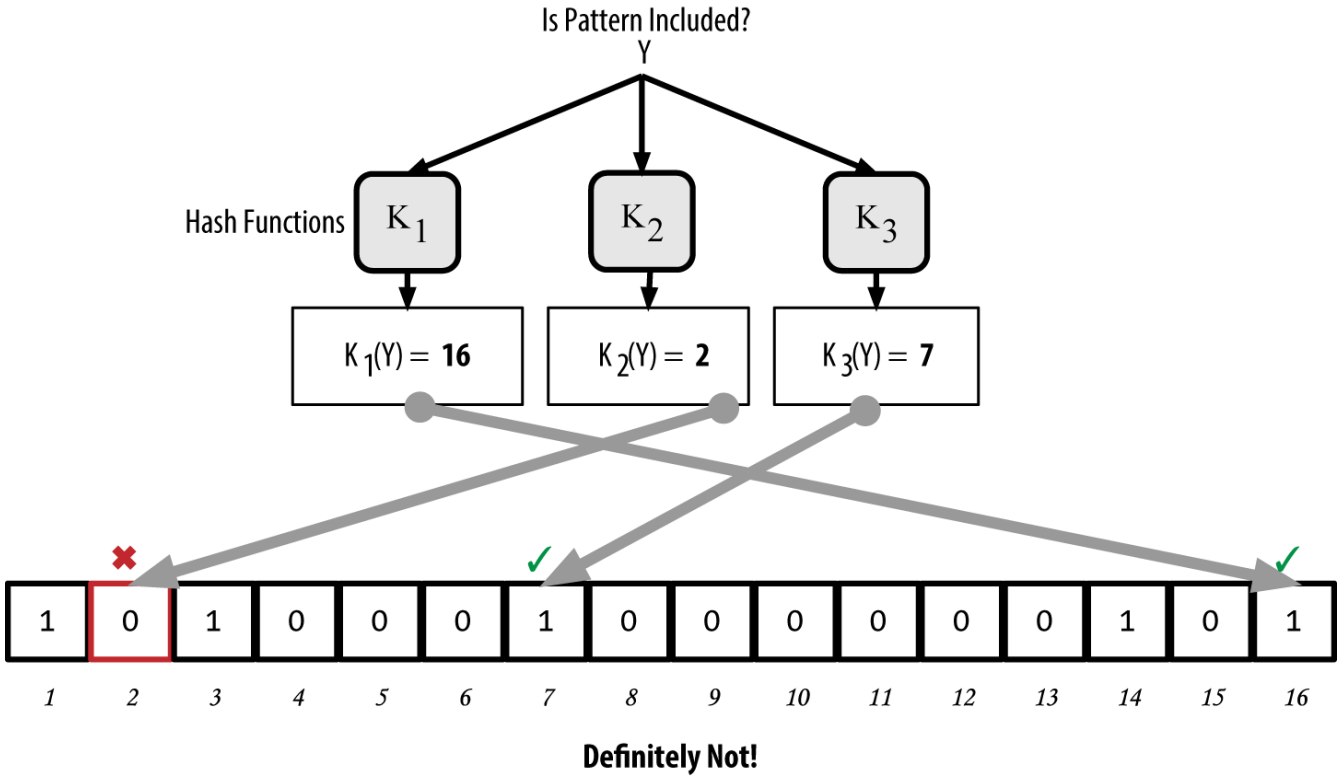


Figure 12. Testing the existence of pattern "Y" in the bloom filter. The result is a definitive negative match, meaning "Definitely Not!"

How SPV Nodes Use Bloom Filters

Bloom filters are used to filter the transactions (and blocks containing them) that an SPV node receives from its peers, selecting only transactions of interest to the SPV node without revealing which addresses or keys it is interested in.

An SPV node will initialize a bloom filter as "empty"; in that state the bloom filter will not match any patterns. The SPV node will then make a list of all the addresses, keys, and hashes that it is interested in. It will do this by extracting the public key hash and script hash and transaction IDs from any UTXO controlled by its wallet. The SPV node then adds each of these to the bloom filter, so that the bloom filter will "match" if these patterns are present in a transaction, without revealing the patterns themselves.

The SPV node will then send a filterload message to the peer, containing the bloom filter to use on the connection. On the peer, bloom filters are checked against each incoming transaction. The full node checks several parts of the transaction against the bloom filter, looking for a match including:

- The transaction ID
- The data components from the locking scripts of each of the transaction outputs (every key and hash in the script)
- Each of the transaction inputs
- Each of the input signature data components (or witness scripts)

By checking against all these components, bloom filters can be used to match public key hashes, scripts, OP_RETURN values, public keys in signatures, or any future component of a smart contract or complex script.

After a filter is established, the peer will then test each transaction's output against the bloom filter. Only transactions that match the filter are sent to the node.

In response to a getdata message from the node, peers will send a merkleblock message that contains only block headers for blocks matching the filter and a merkle path (see [merkle_trees]) for each matching transaction. The peer will then also send tx messages containing the transactions matched by the filter.

As the full node sends transactions to the SPV node, the SPV node discards any false positives and uses the correctly matched transactions to update its UTXO set and wallet balance. As it updates its own view of the UTXO set, it also modifies the bloom filter to match any future transactions referencing the UTXO it just found. The full node then uses the new bloom filter to match new transactions and the whole process repeats.

The node setting the bloom filter can interactively add patterns to the filter by sending a filteradd message. To clear the bloom filter, the node can send a filterclear message. Because it is not possible to remove a pattern from a bloom filter, a node has to clear and resend a new bloom filter if a pattern is no longer desired.

The network protocol and bloom filter mechanism for SPV nodes is defined in BIP-37 (Peer Services).

SPV Nodes and Privacy

Nodes that implement SPV have weaker privacy than a full node. A full node receives all transactions and therefore reveals no information about whether it is using some address in its wallet. An SPV node receives a filtered list of transactions related to the addresses that are in its wallet. As a result, it reduces the privacy of the owner.

Bloom filters are a way to reduce the loss of privacy. Without them, an SPV node would have to explicitly list the addresses it was interested in, creating a serious breach of privacy. However, even with bloom filters, an adversary monitoring the traffic of an SPV client or connected to it directly as a node in the P2P network can collect enough information over time to learn the addresses in the wallet of the SPV client.

Encrypted and Authenticated Connections

Most new users of bitcoin assume that the network communications of a bitcoin node are encrypted. In fact, the original implementation of bitcoin communicates entirely in the clear. While this is not a major privacy concern for full nodes, it is a big problem for SPV nodes.

As a way to increase the privacy and security of the bitcoin P2P network, there are two solutions that provide encryption of the communications: *Tor Transport* and *P2P Authentication and Encryption* with BIP-150/151.

Tor Transport

Tor, which stands for *The Onion Routing network*, is a software project and network that offers encryption and encapsulation of data through randomized network paths that offer anonymity,

untraceability and privacy.

Bitcoin Core offers several configuration options that allow you to run a bitcoin node with its traffic transported over the Tor network. In addition, Bitcoin Core can also offer a Tor hidden service allowing other Tor nodes to connect to your node directly over Tor.

As of Bitcoin Core version 0.12, a node will offer a hidden Tor service automatically if it is able to connect to a local Tor service. If you have Tor installed and the Bitcoin Core process runs as a user with adequate permissions to access the Tor authentication cookie, it should work automatically. Use the debug flag to turn on Bitcoin Core's debugging for the Tor service like this:

```
$ bitcoind --daemon --debug=tor
```

You should see "tor: ADD_ONION successful" in the logs, indicating that Bitcoin Core has added a hidden service to the Tor network.

You can find more instructions on running Bitcoin Core as a Tor hidden service in the Bitcoin Core documentation (*docs/tor.md*) and various online tutorials.

Peer-to-Peer Authentication and Encryption

Two Bitcoin Improvement Proposals, BIP-150 and BIP-151, add support for P2P authentication and encryption in the bitcoin P2P network. These two BIPs define optional services that may be offered by compatible bitcoin nodes. BIP-151 enables negotiated encryption for all communications between two nodes that support BIP-151. BIP-150 offers optional peer authentication that allows nodes to authenticate each other's identity using ECDSA and private keys. BIP-150 requires that prior to authentication the two nodes have established encrypted communications as per BIP-151.

As of February 2021, BIP-150 and BIP-151 are not implemented in Bitcoin Core. However, the two proposals have been implemented by at least one alternative bitcoin client named bcoin.

BIP-150 and BIP-151 allow users to run SPV clients that connect to a trusted full node, using encryption and authentication to protect the privacy of the SPV client.

Additionally, authentication can be used to create networks of trusted bitcoin nodes and prevent Man-in-the-Middle attacks. Finally, P2P encryption, if deployed broadly, would strengthen the resistance of bitcoin to traffic analysis and privacy-eroding surveillance, especially in totalitarian countries where internet use is heavily controlled and monitored.

The standard is defined in BIP-150 (Peer Authentication) and BIP-151 (Peer-to-Peer Communication Encryption).

Transaction Pools

Almost every node on the bitcoin network maintains a temporary list of unconfirmed transactions called the *memory pool*, *mempool*, or *transaction pool*. Nodes use this pool to keep track of transactions that are known to the network but are not yet included in the blockchain. For example, a wallet node will use the transaction pool to track incoming payments to the user's wallet that

have been received on the network but are not yet confirmed.

As transactions are received and verified, they are added to the transaction pool and relayed to the neighboring nodes to propagate on the network.

Some node implementations also maintain a separate pool of orphaned transactions. If a transaction's inputs refer to a transaction that is not yet known, such as a missing parent, the orphan transaction will be stored temporarily in the orphan pool until the parent transaction arrives.

When a transaction is added to the transaction pool, the orphan pool is checked for any orphans that reference this transaction's outputs (its children). Any matching orphans are then validated. If valid, they are removed from the orphan pool and added to the transaction pool, completing the chain that started with the parent transaction. In light of the newly added transaction, which is no longer an orphan, the process is repeated recursively looking for any further descendants, until no more descendants are found. Through this process, the arrival of a parent transaction triggers a cascade reconstruction of an entire chain of interdependent transactions by re-uniting the orphans with their parents all the way down the chain.

Both the transaction pool and orphan pool (where implemented) are stored in local memory and are not saved on persistent storage; rather, they are dynamically populated from incoming network messages. When a node starts, both pools are empty and are gradually populated with new transactions received on the network.

Some implementations of the bitcoin client also maintain an UTXO database or pool, which is the set of all unspent outputs on the blockchain. Bitcoin Core users will find it in the chainstate/ folder of their client's data directory. Although the name "UTXO pool" sounds similar to the transaction pool, it represents a different set of data. Unlike the transaction and orphan pools, the UTXO pool is not initialized empty but instead contains millions of entries of unspent transaction outputs, everything that is unspent from all the way back to the genesis block. The UTXO pool may be housed in local memory or as an indexed database table on persistent storage.

Whereas the transaction and orphan pools represent a single node's local perspective and might vary significantly from node to node depending upon when the node was started or restarted, the UTXO pool represents the emergent consensus of the network and therefore will vary little between nodes. Furthermore, the transaction and orphan pools only contain unconfirmed transactions, while the UTXO pool only contains confirmed outputs.