Consider an account-based ledger. It might be organized by time and contain a bunch of transactions in order. When we add a new transaction and want to check if it's valid, we'd have to scan back all the way until the genesis block to verify that someone has the correct amount of money in their account. As a result, verification is slow. Miners would have to maintain some parallel data structure to speed up these checks, but that's a big requirement.

Bitcoin, instead, is transaction-based, rather than account-based. There's no notion of a "balance" for an individual address. Hash pointers tell you exactly where to look.

Scripts

Output "addresses" are really *scripts*. Scripts let you specify arbitrary conditions necessary to claim money. Input "addresses" are *also* scripts. The two get concatenated and run to verify that the transaction is legitimate. Usually called *scriptSig* ("input", or "unlock") and *scriptPubKey* ("output", or "lock").

Script is not Turing-complete, but that's by design. It makes the scripts very easy to run, computationally inexpensive, and safe. It's stack-based, has some cryptographic support, limits time and memory, and does not allow looping or recursion. There are 256 opcodes, which include arithmetic operations and some cryptography primitives, like a variety of hash functions.

99.9% of the time, the Bitcoin script follows a predictable format: "$< sig > < pubKey >$ OP_DUP OP_HASH160 $< pubKeyHash >$ OP_EQUALVERIFY OP_CHECKSIG". This is just Pay-to-Public-Key. If anything goes wrong during Script execution, there'll be some error or you'll have something left on the stack at the end, in which case you know that it failed.

Roughly, 0.01% are MULTISIG and 0.01% are Pay-to-Script-Hash. The remaining scripts are errors or proof-of-burn.

Multisig

The "OP_CHECKMULTISIG" operator is particularly interesting. It provides built-in support for join signatures, where you can specify n public keys and say that you require t signatures for verification. Supports at most 15 keys.

Proof-of-burn

These scripts are of the form "OP_RETURN $<$ *data* $>$". They burn money, because OP_RETURN always fails.

Should senders specify scripts?

Usually, the person who's sending the money has to execute the script. It's a little odd. Instead, with Pay-to-Script-Hash, you only send the hash of the script that's going to be executed.

Script Applications

Escrow transactions

Alice wants to buy online from Bob. Alice doesn't want to pay until after Bob ships. Bob doesn't want to ship until after Alice pays.

Pick an escrow agent, Judy, and pay to a 2-of-3 MULTISIG requiring two of Alice, Bob, and Judy. Judy can resolve disputes, etc.: for example, if Bob sends bad goods, she can rule with Alice to invalidate the transactions; or if Alice doesn't pay, Judy can still help Bob gets the funds. In addition, there's no way for Judy to steal the money, which is a nice property–she needs to cooperate with either Alice or Bob to move it. Note that if Judy dies or loses her private key, the money can be lost forever.

Green addresses

Alice wants to pay Bob. Bob doesn't want to wait for six verifications of confirmation (i.e., to guard against double-spends), or is offline completely.

You can introduce a trusted third-party, like a bank. Alice shows the bank her credit card and asks for the bank to make a "green payment" to Bob of the form *Pay x to Bob, y to Bank* (where y is the leftover of the coin the bank is spending). The bank should be trusted by Bob and can assume that there's no double-spend.

If the bank ever did double-spend, there'd be a huge credibility loss.

Efficient micro-payments

Alice is a phone customer, and Bob is the phone company. Alice wants to pay Bob for each minute of phone service, but she doesn't want to incur a transaction fee for each small payment.

We start with a MULTISIG transaction that requires Alice and Bob to verify. Alice sends Bob 100 Bitcoin–more than she'd ever want to spend on phone service. Alice uses her phone for a minute and produces a transaction that sends 1 Bitcoin to Bob and 99 to Alice, which is signed only by Alice (awaiting Bob's signature). She continues to send transactions for each minute. In the end, Bob signs the final transaction. He can't sign multiple transactions because it would be a double-spend.

If Bob never signs, the coins are burned. The fix: demand a timed refund transaction that returns her money at time t.

Forks

What happens if we try to upgrade Bitcoin (i.e., the Bitcoin software)? Why would we even want to do that in the first place?

Well, there are a lot of hard-coded limits (e.g., 10-minute average creation time per block, 1M bytes per block, 100M Satoshis per Bitcoin), and over time these may not be the best choices. In particular, there are some significant throughput limits in Bitcoin (turns out to support roughly 7 transactions per second), compared to 2,000-10,000 transactions per second for VISA. Bitcoin clearly cannot handle the volume of transactions that the world wants.

A "hard-forking" change to Bitcoin would be problematic, because nodes with the new software might accept certain blocks that the old nodes dislike. There are two types of incompatibility: the old nodes reject the new blocks, or the new nodes reject the old blocks (the latter results in an orphan block every time an old node mines a new block). This leads to tons of forking in the blockchain, and the old nodes would never catch up. It's impossible to assume that every node would upgrade, or at least upgrade in time.

"Soft forks" are doable: these limit the set of valid transactions or the set of valid blocks. You need the majority of nodes to enforce these new rules, but the old nodes will still approve them, because the now-approved blocks are a subset of the old-approved nodes. There's a risk in that the old nodes might mine now-invalid blocks; this leads to a temporary fork, but they'll recover and get back onto the main chain.

Pay-to-Script-Hash is a classic example of a soft fork. The old nodes will just approve the hash, not run the embedded script, so they still see these now-valid blocks as valid.

Blocks

We bundle transactions together to create a single unit of work for miners and make the blockchain smaller, because all we need are hashes of blocks, rather than of every single transaction.

The blockchain is a combination of two clever data structures:

- A hash chain of blocks.
- A Merkle tree of the transactions in each block. *This allows for logarithmic lookup of transactions within blocks.*

A real Bitcoin block has a header (with "hash" (must start with a large number of zeros to be a valid block, as only a large number of zeros will lead to a sufficiently small output), "ver", "prev_block", etc.) and then a Merkle tree of transactions. The Merkle root hash ("mrkl_root") is included in the block header.

The header is the only thing that's hashed during mining, and the only transaction data that's included in the header is the Merkle root, a single hash.

The **coinbase transaction** is a special transaction within the block. This is the special transaction that creates new Bitcoins as a reward for the miner. Today, it will be 25 BTC, along with the transaction fees included by those adding transactions.

The pointer to this transaction is a null pointer (i.e., a hash of all zeros). There's no previous transaction that's being consumed to create these coins. There's also a coinbase parameter that's included in that transaction where you can include some additional data.

The Bitcoin Network

The Bitcoin network is a P2P network. All nodes are considered equal: there's no master node, no hierarchy, etc. It runs over TCP with a random topology (i.e., nodes are connected randomly) and new nodes can join any time.

There's no explicit way to leave the network. Instead, if you haven't been heard of recently (i.e., in 3 hours), you're considered "gone".

Joining

When you launch a new node, you start with a single message to one node that you know about (this is called the *seed node*) requesting the addresses of all the nodes that this node is aware of. You then send this same message to some of these addresses, iterating as many times as you want until you have a list of peers to connect with.

Flooding

You want to entire network to hear about a transaction that you make. Bitcoin follows a simple transaction propagation (i.e., *flooding*) algorithm. You tell as many people as you can, then when they hear the news, they tell as many people as they can, etc., also known as a *gossip protocol*.

Each node maintains a list of all the transactions they've heard about (that haven't been included in the blockchain). If you try to propagate a transaction and the recipient node has already heard about it, it'll just deny the request, as it can identify transactions very quickly by their hashes.

Before propagating a transaction, you can check that it's valid with the current blockchain. Typically, you also check that the script matches a whitelist, verify that you haven't seen it before, and make sure it doesn't conflict with other transactions you've relayed. (These are not required, which makes it important that every node does its own verifications.)

At different points in time, nodes will have a different view of the world: one node may consider transaction T valid, while others may not. The network can end up in a divided state. But this is fine because the transactions have not been published in the blockchain. In practice, this is a race condition, which is why network position matters.

Block propagation

Nearly identical to transaction propagation, except that the verification protocol is different: in addition to validating the header, nodes are also asked to validate every transaction in the block; and blocks will only be forwarded if they build on that node's current longest chain (avoids forks building up). (Again, however, none of these are *required* by the protocol and it's designed to withstand a lack of checks.)

Propagation time

The average block propagation time is over 30 seconds. The protocol was not designed to be efficient: it was designed to be simple. As a result, the topology of the network *will not be optimized for fast communication.*

For Bitcoin, it's worth giving up some propagation time in return for decentralization and equality of nodes.

Network size

Impossible to measure exactly and as it's changing all the time. Some estimates say around one million IP addresses per month; however, there are only about 5,000-10,000 "full nodes".

Node diversity

Two primary types of nodes.

Fully-validating nodes

These store the entire blockchain.

There's some concern that the number of fully-validating nodes is going down: it's expensive in that you need to stay connected with an active network connection and must store the entire blockchain (20GB).

In addition, you want the entire UTXO set to be in RAM (1GB) so as to verify quickly. This is becoming more difficult over time.

Thin clients

These nodes do not store everything: just block headers (i.e., make sure they were difficult to mine). They request transactions as-needed and trust fully-validating nodes. Their behavior leads to a 1000x cost saving!

Software diversity

Roughly 90% of nodes run "Core Bitcoin" (C++). There are other implementations running successfully in Java (BitcoinJ), Go (btcd), and more languages.

The "Original Satoshi client" was the code released with the first version of Bitcoin. Famously, it was very bizarrely written and difficult to maintain. At this point, it's mostly a historical curiosity.