Bitcoin Overview

Bitcoin Protocol

- Standards
  - Hashes
    - Usually, when a hash is computed within bitcoin, it is computed twice
    - Most of the time SHA-256 hashes are used
    - however RIPEMD-160 is also used when a shorter hash is desirable
      - for example when creating a bitcoin address
  - Merkle Tree
    - Binary trees of hashes
    - bitcoin uses a **double** SHA-256
    - Example block with three transactions a, b, and c:

```
d1 = dhash(a)
d2 = dhash(b)
d3 = dhash(c)
d4 = dhash(c)          # a, b, c are 3. that's an odd number,
                       # so we take the c twice
d5 = dhash(d1 concat d2)
d6 = dhash(d3 concat d4)

d7 = dhash(d5 concat d6)
```

  - Signatures
    - Bitcoin uses Elliptic Curve Digital Signature Algorithm (ECDSA) to sign transactions
    - Public keys (in scripts) are given as either:
      - `04 <x> <y>` where `x` and `y` are 32 byte big-endian integers representing the coordinates of a point on the curve
      - or in compressed form given as `<sign> <x>` where `<sign>` is `0x02` if `y` is even and `0x03` if `y` is odd.
  - Transaction Verification
    - Transactions are cryptographically signed records that reassign ownership of Bitcoins to new addresses
    - Transactions have
      - *inputs* - records which reference the funds from other previous transactions
      - *outputs* - records which determine the new owner of the transferred Bitcoins

            - will be referenced as inputs in future transactions as those funds are respent.
    - Inputs
        - must have a cryptographic digital signature that unlocks the funds from the prior transaction
        - Only the person possessing the appropriate private key is able to create a satisfactory signature
        - this **ensures that funds can only be spent by their owners**
    - Outputs
        - determines which Bitcoin address is the recipient of the funds
        - Most Bitcoin outputs encumber the newly transferred coins with a single ECDSA private key
        - The actual record saved with inputs and outputs isn't necessarily a key, but a *script*
            - Bitcoin uses an interpreted scripting system to determine whether an output's criteria have been satisfied
            - more complex operations are possible
                - outputs that require two ECDSA signatures
                - two-of-three-signature schemes
        - An output that references a single Bitcoin address is a *typical* output
        - The output script specifies what must be provided to unlock the funds later
            - when the time comes in the future to spend the transaction in another input, that input must provide all of the things that satisfy the requirements defined by the original output script
    - Execution
        - In a transaction, the sum of all inputs must be greater than or equal to the sum of all outputs
        - If the inputs exceed the outputs, the difference is considered a transaction fee, and is redeemable by whoever first includes the transaction into the blockchain
    - Coinbase transaction
        - Has no inputs
        - contains outputs totalling the sum of the block reward plus all transaction fees collected from the other transactions in the same block
- Addresses
    - A bitcoin address is the hash of a ECDSA public key

- Common Structures
    - Message Structure

Field Size	Description	Data type	Comments
4	magic	uint32_t	Magic value indicating message origin network, and used to seek to next message when stream state is unknown
12	command	char[12]	ASCII string identifying the packet content, NULL padded (non-NULL padding results in packet rejected)
4	length	uint32_t	Length of payload in number of bytes
4	checksum	uint32_t	First 4 bytes of sha256(sha256(payload))
?	payload	uchar[]	The actual data

    - Variable length integer
    - Variable length string
    - Network address
    - Inventory vectors
        - used for notifying other nodes about objects they have or data which is being requested
    - Block Headers
        - Block headers are sent in a headers packet in response to a getheaders message

Field Size	Description	Data type	Comments
4	version	int32_t	Block version information (note, this is signed)
32	prev_block	char[32]	The hash value of the previous block this particular block references
32	merkle_root	char[32]	The reference to a Merkle tree collection which is a hash of all transactions related to this block
4	timestamp	uint32_t	A timestamp recording when this block was created
4	bits	uint32_t	The calculated difficulty target being used for this block

4	nonce	uint32_t	The nonce used to generate this block… to allow variations of the header and compute different hashes
1	txn_count	var_int	Number of transaction entries, this value is always 0

- Prefilled Transaction
- HeaderAndShortIDs
- Block Transactions Request
  - structure is used to list transaction indexes in a block being requested

Field Name	Type	Size	Encoding	Purpose
blockhash	Binary blob	32 bytes	The output from a double-SHA256 of the block header, as used elsewhere	The blockhash of the block which the transactions being requested are in
indexes_length	CompactSize	1 or 3 bytes	As used to encode array lengths elsewhere	The number of transactions being requested
indexes	List of CompactSizes	1 or 3 bytes*indexes_length	Differentially encoded	The indexes of the transactions being requested in the block

- Block Transactions
  - used to provide some of the transactions in a block, as requested

Field Name	Type	Size	Encoding	Purpose
blockhash	Binary blob	32 bytes	The output from a double-SHA256 of the block header, as used elsewhere	The blockhash of the block which the transactions being provided are in
transactions_length	CompactSize	1 or 3 bytes	As used to encode array lengths elsewhere	The number of transactions provided
transactions	List of Transactions	variable	As encoded in tx messages	The transactions provided

- Message Types
  - version
    - When a node creates an outgoing connection it will immediately advertise its version
    - The remote node will respond with its version
    - No further communication is possible until the nodes have exchanges versions
  - verack
    - The *verack* message is sent in reply to the *version* message
  - addr
    - provides information on known nodes of the network
  - inv
    - Allows a node to advertise its knowledge of one or more objects
    - can be received unsolicited, or in reply to *getblocks*

Field Size	***Description***	***Data type***	***Comments***
?	*count*	*var_int*	*Number of inventory entries*
36x?	*inventory*	*inv_vect[]*	*Inventory vectors*

  - getdata
    - *getdata* is used in response to *inv*, to retrieve the content of a specific object
    - usually sent after receiving an *inv* packet, after filtering known elements
  - notfound
    - notfound is a response to a getdata, sent if any requested data items could not be relayed
    - for example, because the requested transaction was not in the memory pool or relay set
  - getblocks
    - Returns an *inv* packet
      - containing the list of blocks starting right after the last known hash in the block locator object, up to hash_stop or 500 blocks, whichever comes first
  - getheaders

        - Returns a *headers* packet
            - containing the headers of blocks starting right after the last known hash in the block locator object, up to hash_stop or 2000 blocks, whichever comes first
    - tx
        - Bitcoin transaction
        - In reply to getdata

transaction structure:

<table>
<tr><th>Field Size</th><th>Description</th><th>Data type</th><th>Comments</th></tr>
<tr><td>4</td><td>version</td><td>int32_t</td><td>Transaction data format version (note, this is signed)</td></tr>
<tr><td>1+</td><td>tx_in count</td><td>var_int</td><td>Number of Transaction inputs</td></tr>
<tr><td>41+</td><td>tx_in</td><td>tx_in[]</td><td>A list of 1 or more transaction inputs or sources for coins</td></tr>
<tr><td>1+</td><td>tx_out count</td><td>var_int</td><td>Number of Transaction outputs</td></tr>
<tr><td>9+</td><td>tx_out</td><td>tx_out[]</td><td>A list of 1 or more transaction outputs or destinations for coins</td></tr>
<tr><td>4</td><td>lock_time</td><td>uint32_t</td><td>The block number or timestamp at which this transaction is unlocked:

<table><tr><th>Value</th><th>Description</th></tr><tr><td>0</td><td>Not locked</td></tr><tr><td>< 500000000</td><td>Block number at which this transaction is unlocked</td></tr><tr><td>>= 500000000</td><td>UNIX timestamp at which this transaction is unlocked</td></tr></table>If all TxIn inputs have final (0xffffffff) sequence numbers then lock_time is irrelevant. Otherwise, the transaction may not be added to a block until after lock_time (see NLockTime).</td></tr>
</table>

transaction input (tx_in) structure:

Field Size	Description	Data type	Comments
36	previous_output	**outpoint**	The previous output transaction reference, as an OutPoint structure
1+	script length	var_int	The length of the signature script
?	signature script	uchar[]	Computational Script for confirming transaction authorization
4	sequence	uint32_t	Transaction version as defined by the sender. Intended for "replacement" of transactions when information is updated before inclusion into a block.

transaction input → outpoint structure:

Field Size	Description	Data type	Comments
32	hash	char[32]	The hash of the referenced transaction.
4	index	uint32_t	The index of the specific output in the transaction. The first output is 0, etc.

transaction output (tx_out) structure:

Field Size	Description	Data type	Comments
8	value	int64_t	Transaction Value
1+	pk_script length	var_int	Length of the pk_script

?	pk_script	uchar[]	Usually contains the public key as a Bitcoin script setting up conditions to claim this output.

- block
  - **block** message is sent in response to a getdata message which requests transaction information from a block hash

Field Size	Description	Data type	Comments
4	version	int32_t	Block version information (note, this is signed)
32	prev_block	char[32]	The hash value of the previous block this particular block references
32	merkle_root	char[32]	The reference to a Merkle tree collection which is a hash of all transactions related to this block
4	timestamp	uint32_t	A Unix timestamp recording when this block was created (Currently limited to dates before the year 2106!)
4	bits	uint32_t	The calculated difficulty target being used for this block
4	nonce	uint32_t	The nonce used to generate this block… to allow variations of the header and compute different hashes
?	txn_count	var_int	Number of transaction entries
?	txns	tx[]	Block transactions, in format of "tx" command

- headers
  - The headers packet return block headers in response to a getheaders packet
- getaddr
  - The getaddr message sends a request to a node asking for information about known active peers to help with finding potential nodes in the network
  - The response to receiving this message is to transmit one or more addr messages with one or more peers from a database of known active peers
- mempool

        - The mempool message sends a request to a node asking for information about transactions it has verified but which have not yet confirmed
        - The response to receiving this message is an inv message containing the transaction hashes for all the transactions in the node's mempool.
    - ping
        - The *ping* message is sent primarily to confirm that the TCP/IP connection is still valid.
        - An error in transmission is presumed to be a closed connection and the address is removed as a current peer.
    - pong
        - The *pong* message is sent in response to a *ping* message
    - reject
        - The *reject* message is sent when messages are rejected
    - filterload, filteradd, filterclear, merkleblock
    - alert
    - sendheaders
    - feefilter
    - sendsmpct
    - getblocktxn
    - blocktxn

Bitcoin vs. Ethereum

So I've been going through the protocol documentation for bitcoin and have been taking some notes. I think I have a much better understand of how bitcoin works technically and how bitcoin-abe interacts with it. So basically, Abe parses the blockchain into chains, blocks, addresses, transactions. But all of this is very specific to Bitcoin structure and api.

For example, a block .dat file (i.e. blk001.dat) contains the block version, the hash value of the previous block this particular block references, the reference to a Merkle tree collection which is a hash of all transactions related to this block, a Unix timestamp recording when this block was created, the calculated difficulty target being used for this block, the nonce used to generate this block, the number of transactions in the block, and an array containing those transactions.

All of this info is displayed in the bitcoin-abe interface. So it makes sense why bitcoin-abe doesn't work with many other currencies like Ethereum, which is conceptually the same. It is also a programmable block chain with similar hashing protocol, but it is structurally different and cannot be parsed by bitcoin-abe.

But bitcoin-abe does work with other currencies like Namecoin. This is because they have the same structure as bitcoin. Most of them in fact, were forks from the bitcoin software!

Blockchain scalability

There seem to be three major issues with blockchain scalability.

1. Centralization as the blockchain grows.

   As more and more blocks are added to the blockchain, the more the requirements become for storage, bandwidth, and computational power that must be spent by "**full nodes**" in the network, leading to a risk of much higher centralization if the blockchain becomes large enough that only a few nodes are able to process a block.

2. Bitcoin has a hard limit of 1MB per block (~10 minutes)

   Bitcoin has a hard limit of 1MB per block, removing this limit in order to change block size would require a hard fork of the blockchain. A hard fork means making a radical change in protocol so that the nodes following the previous protocol would become invalid. It is a permanent divergence from the previous version of the blockchain. This would be very disruptive to the bitcoin blockchain for several reasons.

   First, transactions are defined by the complete blockchain. Each transaction is made up of *inputs* - records which reference the funds from other previous transactions, and *outputs* - records which determine the new owner of the transferred Bitcoins, which will be referenced as inputs in future transactions as those funds are re-spent. So transactions need knowledge of previous transactions and blocks; this will present a large problem if a hard fork occurs because all of these previous transactions will become invalidated.

   Second, transaction verification by miners requires knowledge of the entire block chain. In a transaction, the sum of all inputs must be greater than or equal to the sum of all outputs. A transaction input structure contains a field with a **outpoint** structure, which is the previous output transaction reference. When a new transaction appears, a miner references the hash of this previous transaction block verifies it, and continues to verify all the way back through the chain.

There are several protocols that require knowledge of the full blockchain. All of these protocols would have to be changed or modified or force compatibility between the old block chain and the new block chain after a hard fork.

3. Transaction fees might have to increase as the network grows