BLOCKCHAIN 1.0 –

BITCOIN AND CRYPTOCURRENCY

BLOCKCHAIN 1.0

BITCOIN AND CRYPTOCURRENCY : Block Hash - structure of block – syntax , structures, and validation - transaction life cycle- transaction types – Hash computation and Merkle Hash Tree -Bitcoin and importance- Creation of coins–Bitcoin P2P Network-, Bitcoin protocols - Mining strategy and rewards – PoW and PoS – Difficulty, hash rate– Wallets- Double spending – forking - Token, Coinbase - practice on MTH

What is Blockchain?

- A Platform for executing transactional services
- Spanned over multiple organizations or individuals who may not (need not) **trust** each other
- An append-only shared ledger of digitally signed and encrypted transactions replicated across a network of peer nodes

The Block in a Blockchain – Securing Data Cryptographically

- Digitally signed and encrypted transactions **"verified" by peers**
- **Cryptographic security** – Ensures that participants can only view information on the ledger that they are authorized to see

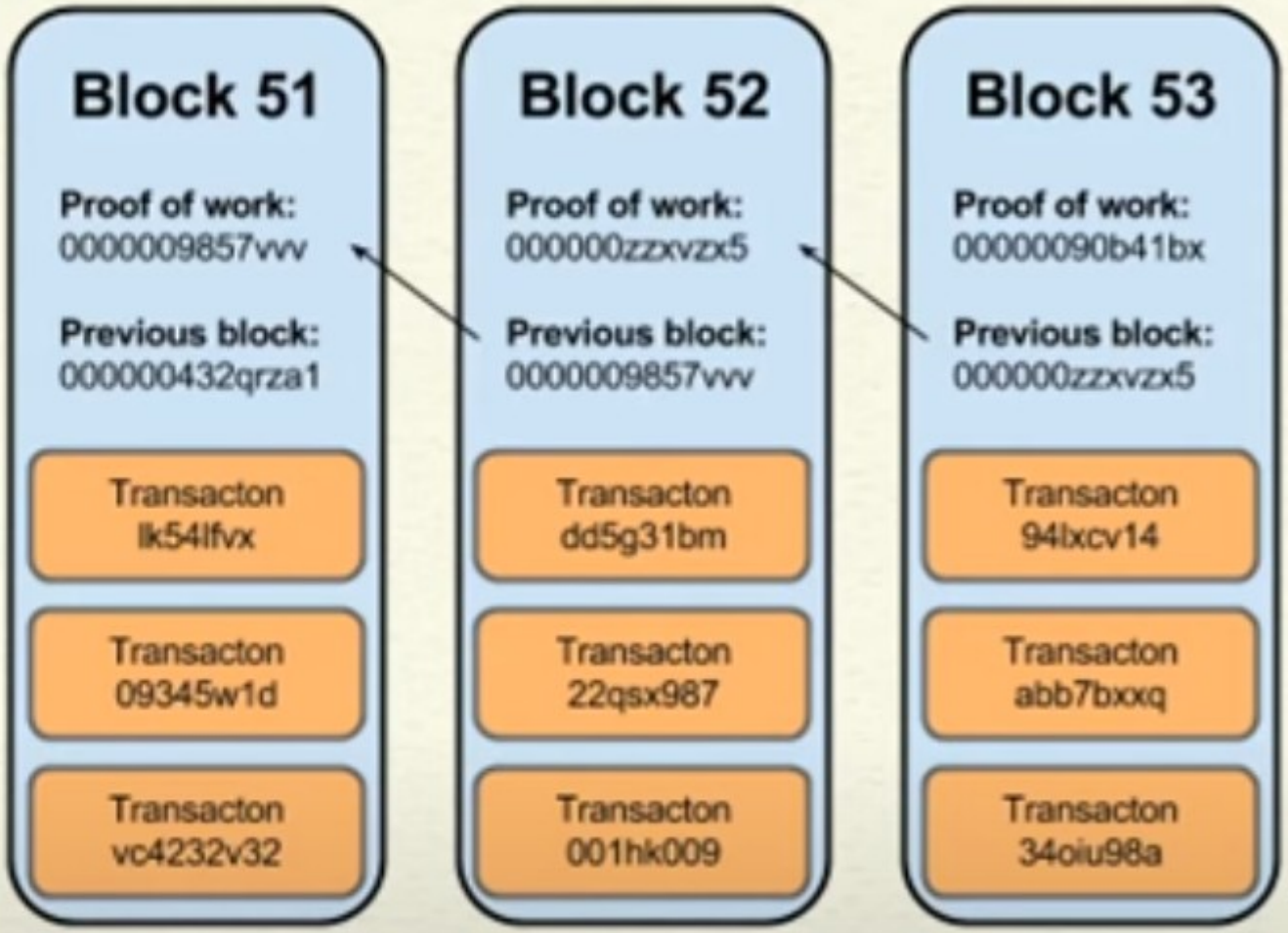
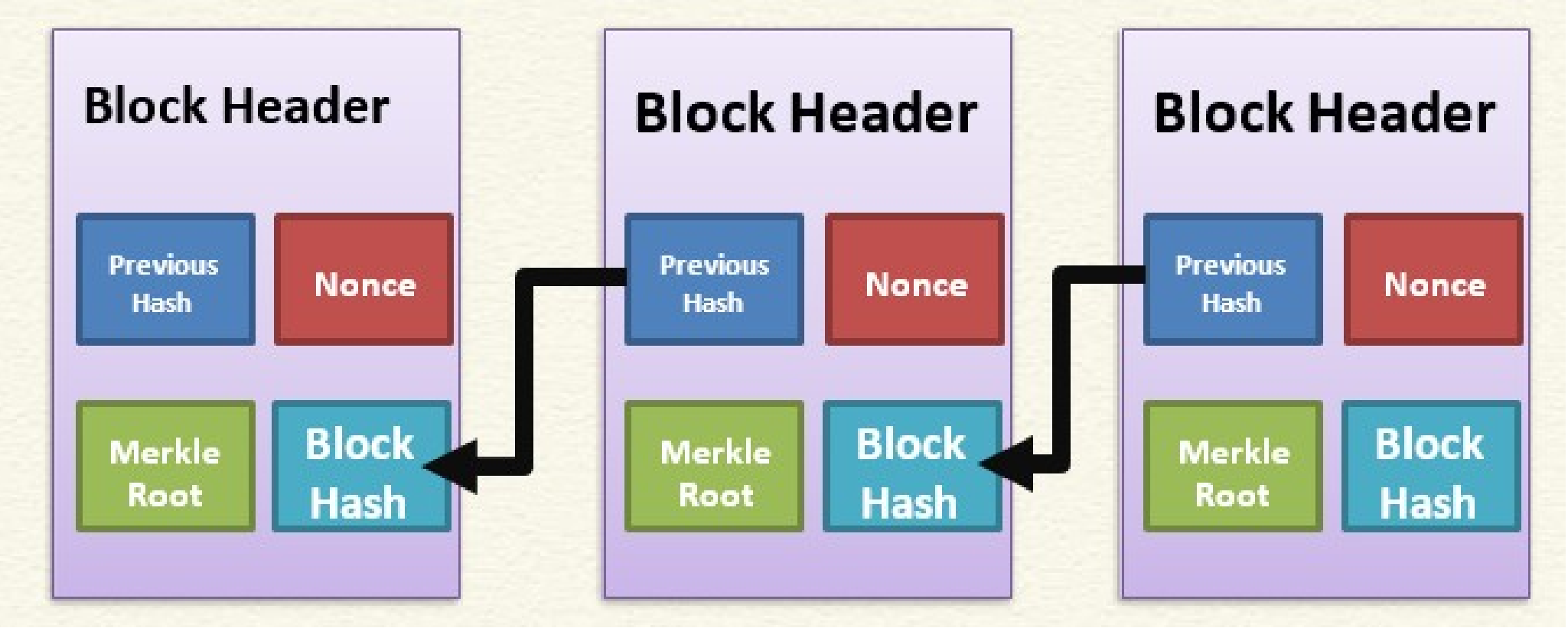


Image source: http://dataconomy.com/

Structure of a Block

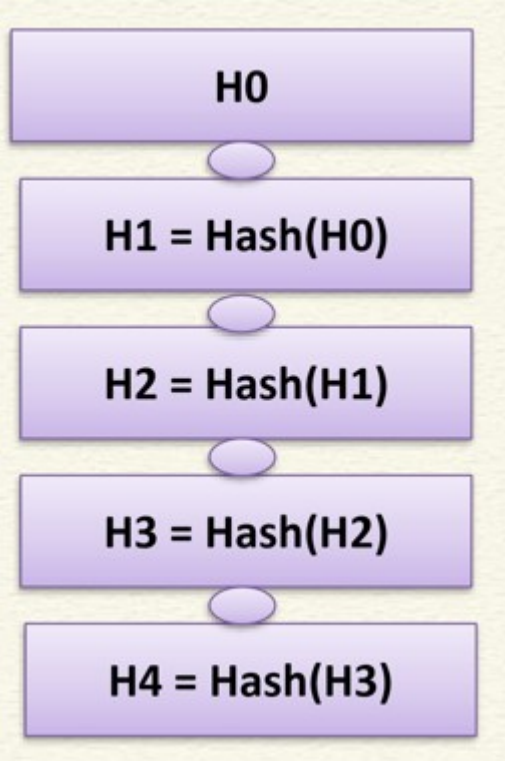
- A block is a **container data structure** that contains a series of transactions
- **In Bitcoin** A block may contain more than 500 transactions on average, the average size of a block is around 1 MB (an upper bound proposed by Satoshi Nakamoto in 2010
- May grow up to 8 MB or sometime higher (several conflicting views on this!!)
- Larger blocks can help in processing large number of transactions in one go.
- But longer time for verification and propagation

Block Generation



Block Header - Bitcoin

- Metadata about a block –
  (1) Previous block hash
  (2) Mining statistics used to construct the block
  (3) Merkle tree root
- Previous block hash: Every block inherits from the previous block we use previous block's hash to create the new block's hash – make the blockchain tamper proof



Block Header - Bitcoin

- Mining – the mechanism to generate the hash
  - The mechanism needs to be complicated enough, to make the blockchain tamper proof
  - Bitcoin Mining: H_k = Hash(H_{k-1} || T || Nonce || Something more)
  - Find the nonce such that H_k has certain predefined complexity (number of zeros at the prefix)
- The header contains mining statistics – timestamp, nonce and difficulty

Block Hash

- A *block hash* (or block ID) is a **unique reference** for a block in the blockchain.
- Every block hash is unique and is determined by the contents of the block. You can therefore use the block hash to search for a specific block in a blockchain explorer.

For example:

- Most Recent Block: 00000000000000000002310849bb724f718b64cd9d56c76f485a2f1948cc15df
- Block 123,456: 0000000000002917ed80650c6174aac8dfc46f5fe36480aaef682ff6cd83c3ca
- Genesis Block: 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
- Notice that all block hashes begin with a **bunch of zeros**. This is because for a block to be added to the blockchain, a miner must get a hash for their block below the current target value. And if the block hash is *below* this target value, then the block hash is naturally going to have a bunch of zeros at the start.

Block Generation Cost

- Energy efficiency ~0.098 J/GH = ~100 J/TH
- ASIC Hardware for bitcoin can perform about 750 TH/s
- Hash rate approx. 120M TH/s!! Many actually go waste Λ
- Network consumes about 80 TW-hours of electricity annually. Figures vary between sources and are some form of estimates
- Average household in Germany of four people consumes approx. 4,000 KW-hours of electricity per year.
- Can power about 20,000 households
- Concept of Pooling is used (https://btc.com/)
- What ensures tamperproof operation in terms of honest nodes?

Blockchain Replicas

- Every peer in a Blockchain network maintains a local copy of the Blockchain.
- Size is just about 351 GB
- As a new user joins the network, she can get the whole copy

Requirements

- All the replicas need to be updated with the last mined block
- All the replicas need to be consistent– the copies of the Blockchain at different peers need to be exactly similar

Validation:

Initial Block Download (IBD):

- When a node joins the network, it downloads the entire blockchain and validates each block against the network rules.

Regular Block Validation:

- Nodes validate each new block received, checking: The block's structure and format.
- The validity of transactions within the block.
- Whether the block's hash meets the current difficulty target.
- The validity of the previous block's hash.

Transactions in a Block

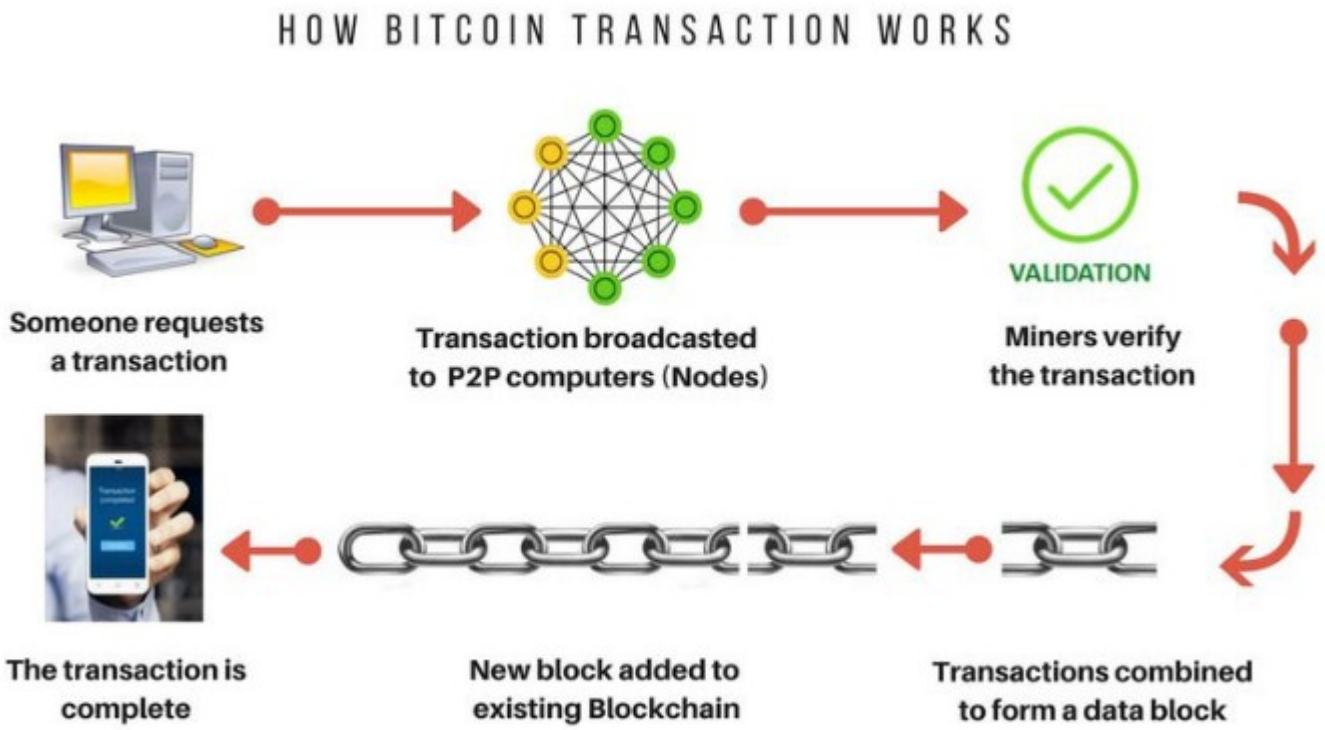
- Transactions are organized as a Merkle Tree. The Merkle Root is used to construct the block hash
- If you change a transaction, you need to change all the subsequent block hashes
- The difficulty of the mining algorithm determines the toughness of tampering with a block in a blockchain

The transaction life cycle

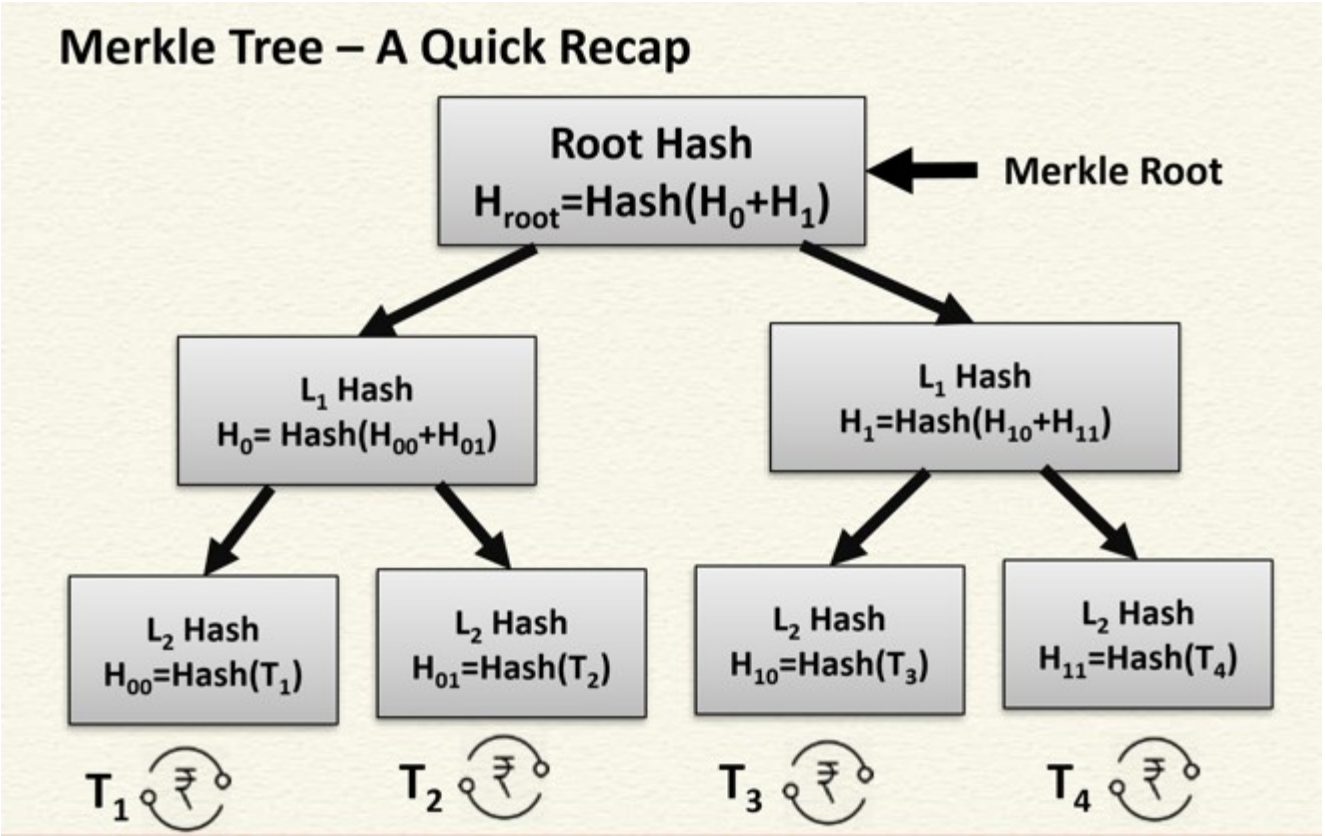
1. User/sender sends a transaction using wallet software or some other interface.
2. The wallet software signs the transaction using the sender's private key.
3. The transaction is broadcasted to the Bitcoin network using a flooding algorithm.
4. Mining nodes (miners) who are listening for the transactions verify and include this transaction in the next block to be mined. Just before the transaction are placed in the block they are placed in a special memory buffer called **transaction pool**.
5. Mining starts, which is a process by which the blockchain is secured and new coins are generated as a reward for the miners who spend appropriate computational resources.

6. Once a miner solves the PoW problem it broadcasts the newly mined block to the network.
7. The nodes verify the block and propagate the block further, and confirmations start to generate.
8. Finally, the confirmations start to appear in the receiver's wallet and after approximately six confirmations, the transaction is considered finalized and confirmed. However, six is just a recommended number; the transaction can be considered final even after the first confirmation. The key idea behind waiting for six confirmations is that the probability of double spending is virtually eliminated after six confirmations

Transaction life cycle



Merkle Tree


Merkle Tree – A Quick Recap
Root Hash
$H_{root}=Hash(H_0+H_1)$
Merkle Root
L_1 Hash
$H_0= Hash(H_{00}+H_{01})$
L_1 Hash
$H_1=Hash(H_{10}+H_{11})$
L_2 Hash
$H_{00}=Hash(T_1)$
L_2 Hash
$H_{01}=Hash(T_2)$
L_2 Hash
$H_{10}=Hash(T_3)$
L_2 Hash
$H_{11}=Hash(T_4)$
T_1
T_2
T_3
T_4

Merkle Tree

- A Merkle tree in Bitcoin is a cryptographic data structure that efficiently summarizes all transactions within a block.
- It acts like a digital fingerprint for the block's transactions, allowing for quick verification of data integrity.
- This summary, called the Merkle root, is stored in the block header, enabling light clients to verify transactions without needing to download the entire blockchain.

Working of Merkle Tree

1. Hashing:

- Each transaction in a block is first hashed using a cryptographic hash function (like SHA-256 in Bitcoin).

2. Pairing and Hashing:

- These transaction hashes are then paired up, and each pair is hashed again. This process continues, with the resulting hashes being paired and hashed until only one hash remains – the Merkle root.

3. Merkle Root:

- The Merkle root represents a concise summary of all transactions in the block.

4. Verification:

- Light clients can download only the Merkle root and a few transaction hashes (a Merkle proof) to verify if a specific transaction is included in a block, without needing to download the entire block.

Transaction fee

- Transaction fees are charged by the miners. The fee charged is dependent upon **the size and weight of the transaction**.
- Transaction fees are calculated by subtracting the sum of the inputs and the sum of the outputs.
- A formula can be used***: fee = sum(inputs) - sum(outputs).***
- The fees are used as an **incentive for miners** to encourage them to include a user transaction in the block the miners are creating.
- All transactions end up in the memory pool, from where miners pick up transactions based on their priority to include them in the proposed block.
- From a transaction fee point of view, a transaction with a higher fee will be picked up sooner by the miners.
- There are different rules based on which fee is calculated for various types of actions, such as sending transactions, inclusion in blocks, and relaying by nodes.

Transaction fee

- Fees are not fixed by the Bitcoin protocol and are not mandatory; even a transaction with no fee will be processed in due course but may take a very long time.
- This is however no longer practical due to the high volume of transactions and competing investors on the Bitcoin network, therefore it is advisable to provide a fee always.
- The time for **transaction confirmation usually ranges from 10 minutes to over 12 hours** in some cases.
- Transaction time is dependent on transaction fees and network activity.

  If the network is very busy then transactions will take longer to process and if you pay a higher fee then your transaction is more likely to be picked by miners first due to additional incentive of the higher fee.
- **Transaction pools :** memory pools, these pools are basically created in local memory (computer RAM) by nodes in order to maintain a temporary list of transactions that are not yet confirmed in a block. Transactions are included in a block **after passing verification** and based on their priority.

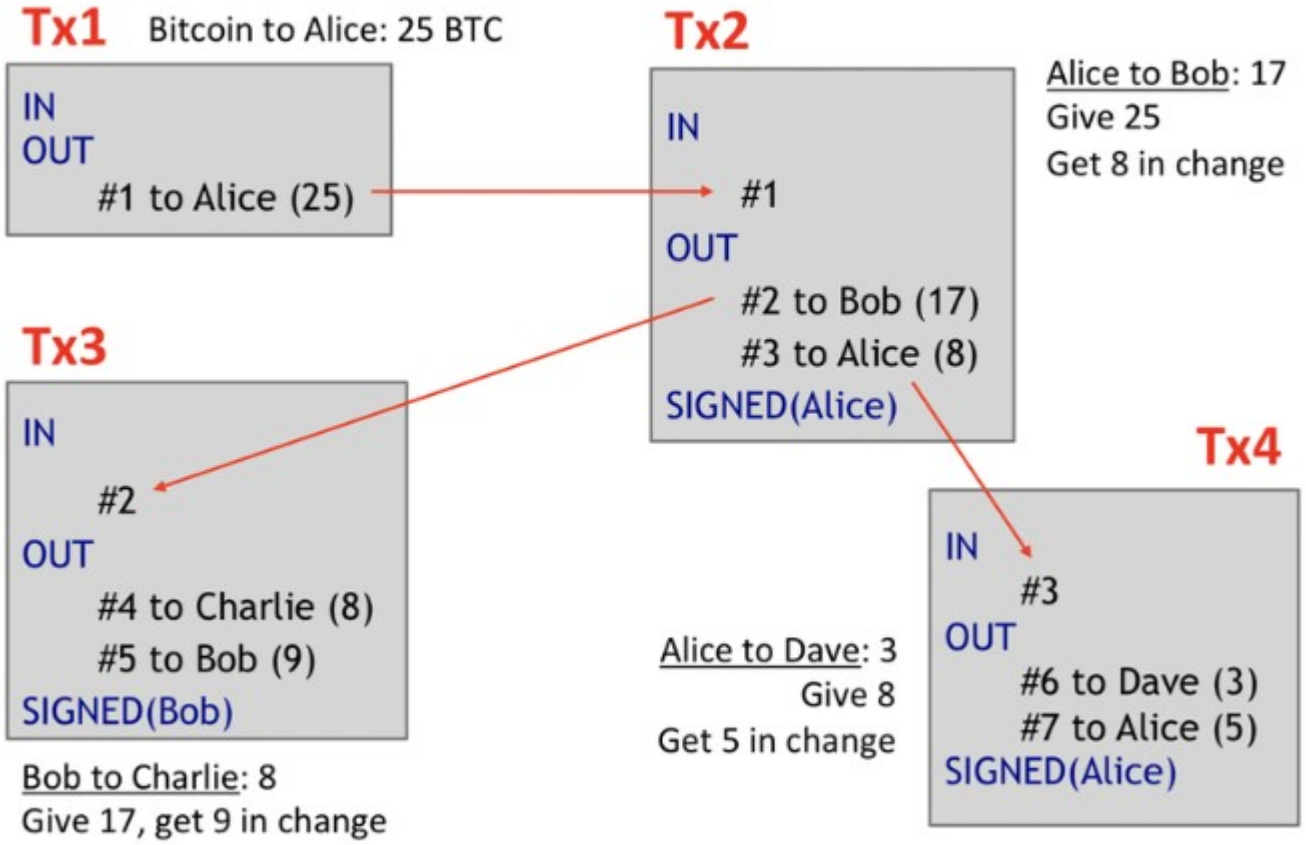
The transaction structure

Field	Size	Description
Version Number	4 bytes	Used to specify rules to be used by the miners and nodes for transaction processing.
input counter	1 bytes-9bytes	The number of inputs included in the transaction.
list of inputs	variable	Each input is composed of several fields, including Previous transaction hash, Previous Txout-index, Txin-script length, Txinscript,and optional sequence number. **The first transaction in a block** is also called a **coinbase transaction**. It specifies one or more transaction inputs.
out-counter	1 bytes-9bytes	A positive integer representing the number of outputs.
list of outputs	variable	Outputs included in the transaction.
lock_time	4 bytes	This defines the earliest time when a transaction becomes valid. It is either a Unix timestamp or a block number.

The transaction structure

- A transaction at a high level contains **metadata, inputs, and outputs**.
- Transactions are combined to create a Block
- **Metadata:** contains the size of the transaction, the number of inputs and outputs, the hash of the transaction, and a lock_time field. Every transaction has a prefix specifying the **version number**.
- **Inputs:** Generally, each input spends a previous output. Each output is considered an **Unspent Transaction Output (UTXO)** until an input consumes it.
- **Outputs:** Outputs have only two fields, and they contain instructions for the sending of bitcoins. The first field contains the amount of Satoshis, whereas the second field is a locking script that contains the conditions that need to be met in order for the output to be spent. More information on transaction spending using locking and unlocking scripts and producing outputs is discussed later in this section.
- **Verification:** Verification is performed using bitcoin's scripting language.

Unspent Transaction Output (UTXO)

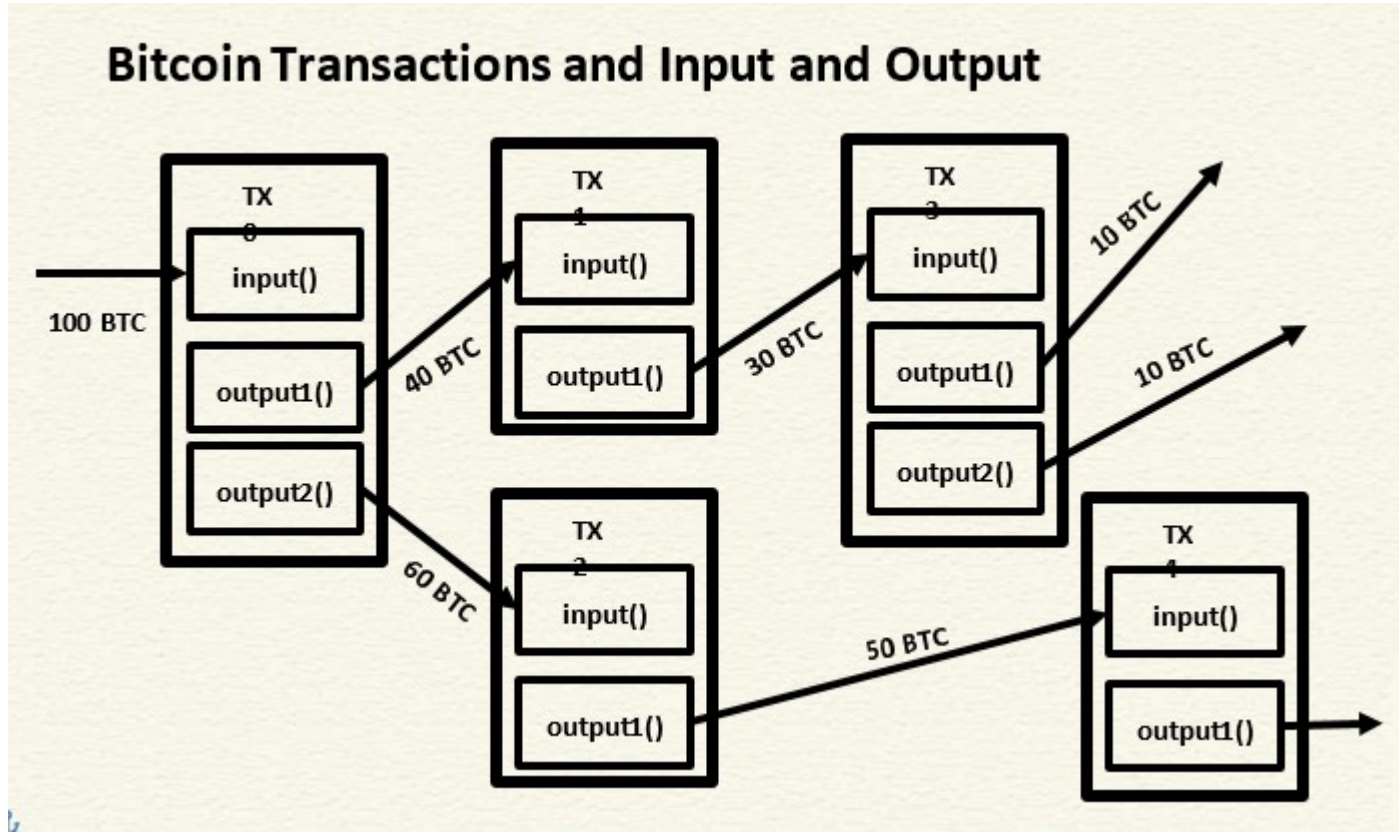


Inputs

Field	Size	Description
Transaction hash	32 bytes	This is the hash of the previous transaction with UTXO.
Output index	4 bytes	This is the previous transactions output index, that is, UTXO to be spent.
Script length	1-9 bytes	This is the size of the unlocking script.
Unlocking script	Variable	Input script (`ScriptSig`) which satisfies the requirements of the locking script.
Sequence number	4 bytes	Usually disabled or contains lock time. Disabled is represented by `'0xFFFFFFFF'`.

Outputs

Field	Size	Description
Value	8 bytes	Total number in positive integers of Satoshis to be transferred
Script size	1-9 bytes	Size of the locking script
Locking script	Variable	Output script (`ScriptPubKey`)


Bitcoin Transactions and Input and Output
TX 0
input()
output1()
output2()
100 BTC
40 BTC
60 BTC
TX 1
input()
output1()
TX 2
input()
output1()
30 BTC
TX 3
input()
output1()
output2()
10 BTC
10 BTC
50 BTC
TX 4
input()
output1()

Verification

- Verification is performed using Bitcoin's scripting language.
- **The script language :** Bitcoin uses a simple stack-based language called script
- describe how bitcoins can be spent and transferred.
- This scripting language is based on a Forth programming language like syntax and uses a reverse polish notation in which every operand is followed by its operators.
- It is evaluated from the left to the right using a Last In, First Out (LIFO) stack.
- **Not Turing Complete** (no loops)
- Scripts use various opcodes or instructions to define their operation.

Bitcoin Script Instructions

- Total 256 opcodes (15 disabled, 75 reserved)
- Arithmetic operations
- if-then conditions
- Logical operators
- Data handling (like OP_DUP)
- Cryptographic operations
- Hash functions
- Signature verification
- Multi-signature verification

Bitcoin Scripts

- Bitcoin scripts are small programs written in a Forth-like, stack-based language that dictate how Bitcoin transactions are validated and how funds can be spent.
- They are used to define the conditions under which an output (an unspent transaction) can be used as input in a new transaction.
- These scripts are fundamental to Bitcoin's functionality, enabling features like multi-signature transactions and more complex transaction logic.

Stack-based language:

- Bitcoin Script is a stack-based language, meaning it operates by pushing data onto and popping data off a stack.

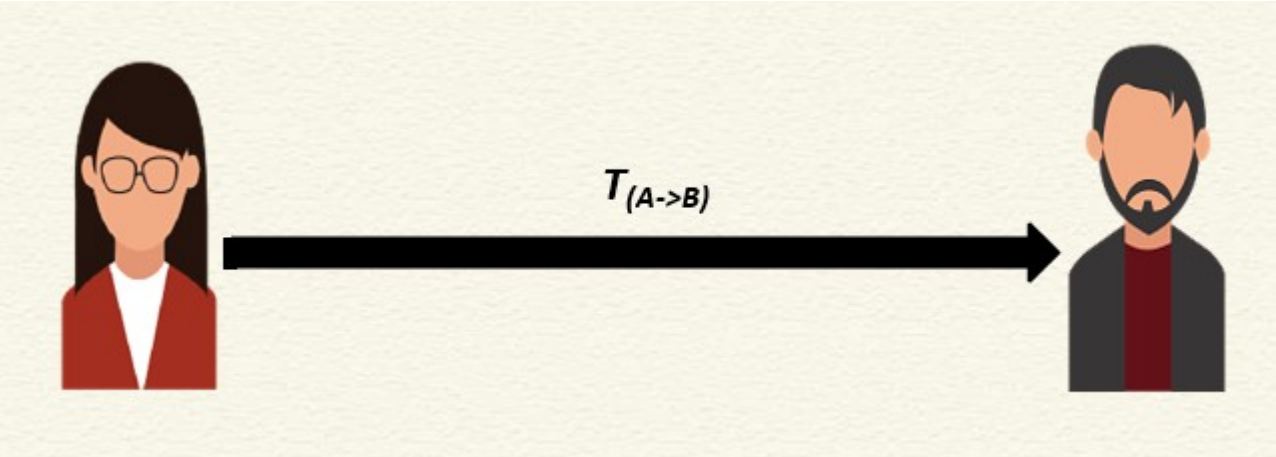
Limited and non-Turing complete:

- It's intentionally designed to be limited and not Turing-complete, meaning it cannot perform complex computations or loops.
- This limitation is crucial for security, preventing infinite loops and ensuring predictable transaction processing.

Locking and unlocking scripts:

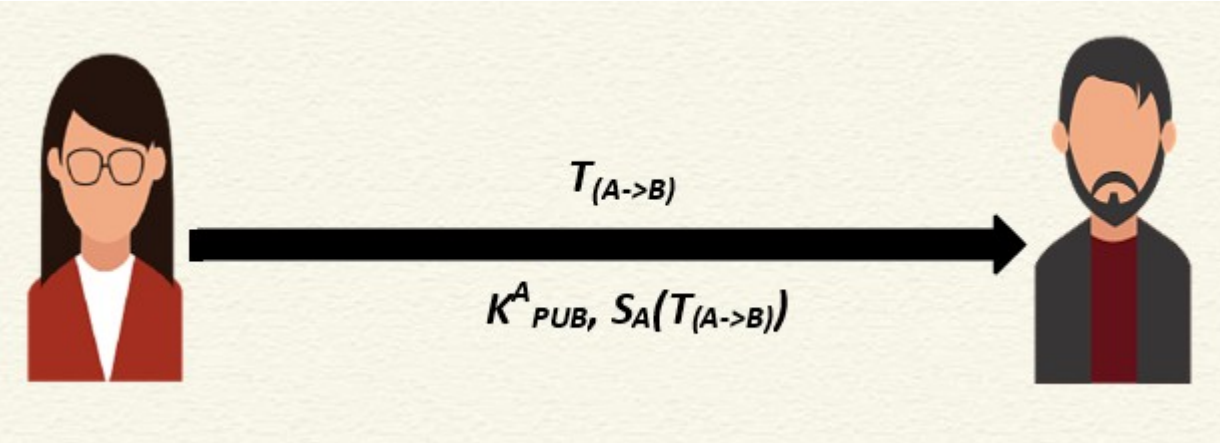
- Each transaction output (UTXO) has a locking script (ScriptPubKey) that specifies the conditions for spending those funds.
- To spend those funds, an unlocking script (ScriptSig or Witness) must be provided, which satisfies the conditions set by the locking script.

Bitcoin Scripts – A Simple Example



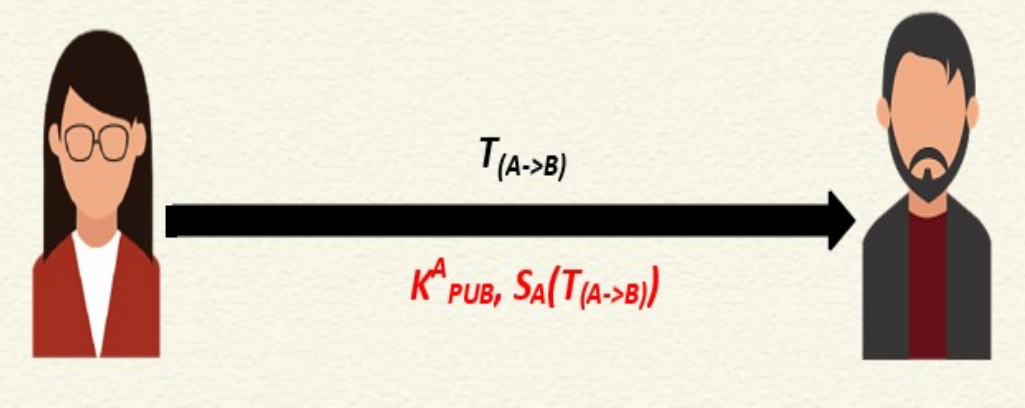
- How Bob will verify that the transaction is actually originated from Alice?

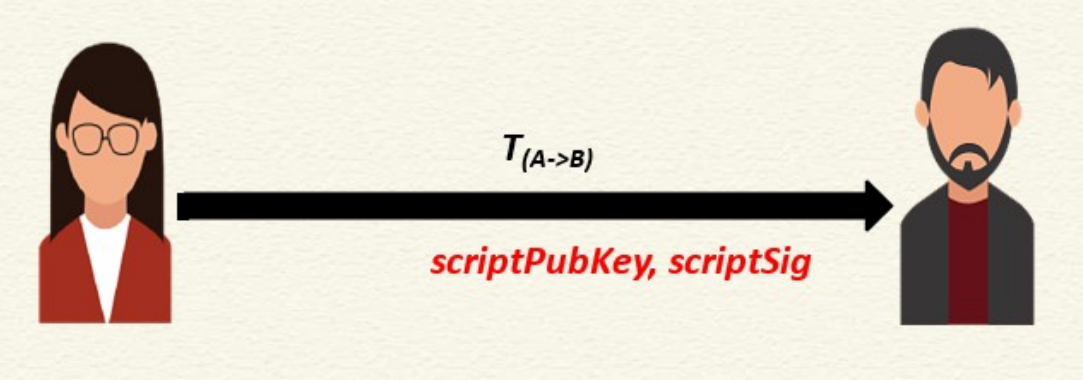
Bitcoin Scripts – A Simple Example



Send the public key of Alice along with the signature -> Bob can verify this

Bitcoin Scripts – A Simple Example





- Bitcoin indeed transfers scripts instead of the signature and the public key

Bob can spend the bitcoin only if both the scripts return TRUE after execution

Bitcoin Scripts

- Simple, compact, stack-based and processed left to right
  - FORTH like language
- Not Turing Complete (no loops)
  - Halting problem is not there

Bitcoin Scripts

- With every transaction Bob must provide
  - A public key that, when hashed, yields the address of Bob embedded in the script
  - A signature to provide ownership of the private key corresponding to the public key of Bob

Verification

- A transaction script is evaluated by combining ***ScriptSig*** and ***ScriptPubKey***.
- ***ScriptSig*** is the unlocking script, whereas ***ScriptPubKey*** is the locking script.
- how a transaction to be spent is evaluated?

1. it is unlocked and then it is spent
2. *ScriptSig* is provided by the user who wishes to unlock the transaction
3. S*criptPubkey* is part of the transaction output and specifies the conditions that need to be fulfilled in order to spend the output
4. In other words, outputs are locked by S*criptPubKey* that contains the conditions, when met will unlock the output, and coins can then be redeemed

Commonly used opcodes

Opcode	Description
OP_CHECKSIG	This takes a public key and signature and validates the signature of the hash of the transaction. If it matches, then TRUE is pushed onto the stack; otherwise, FALSE is pushed.
OP_EQUAL	This returns 1 if the inputs are exactly equal; otherwise, 0 is returned.
OP_DUP	This duplicates the top item in the stack
OP_HASH160	The input is hashed twice, first with SHA-256 and then with RIPEMD-160.
OP_VERIFY	This marks the transaction as invalid if the top stack value is not true.
OP_EQUALVERIFY	This is the same as OP_EQUAL, but it runs OP_VERIFY afterwards.
OP_CHECKMULTISIG	This takes the first signature and compares it against each public key until a match is found and repeats this process until all signatures are checked. If all signatures turn out to be valid, then a value of 1 is returned as a result; otherwise, 0 is returned.

Verification

- There are various scripts available in Bitcoin to handle the value transfer from the source to the destination.
- These scripts range from very simple to quite complex depending upon the requirements of the transaction.
- Standard transactions are evaluated using **IsStandard() and IsStandardTx()** tests and only standard transactions that pass the test are generally allowed to be mined or broadcasted on the Bitcoin network.
- However, nonstandard transactions are valid and allowed on the network.

Types of transactions

- **Pay to Public Key Hash (P2PKH):** P2PKH is the most commonly used transaction type and is used to send transactions to the bitcoin addresses. The format of the transaction is shown as follows:

```
ScriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
ScriptSig: <sig> <pubKey>
```

The ScriptPubKey and ScriptSig parameters are concatenated together and executed.

- **Pay to Script Hash (P2SH):** P2SH is used in order to send transactions to a script hash (that is, the addresses starting with 3) and was standardized in Bitcoin Improvement Proposal (BIP16). In addition to passing the script, the redeem script is also evaluated and must be valid.

```
ScriptPubKey: OP_HASH160 <redeemScriptHash> OP_EQUAL
ScriptSig: [<sig>...<sign>] <redeemScript>
```

Types of transactions

- **MultiSig (Pay to MultiSig): M-of-N MultiSig** transaction script is a complex type of script where it is possible to construct a script that required multiple signatures to be valid in order to redeem a transaction. Various complex transactions such as escrow and deposits can be built using this script. The template is shown here:

```
ScriptPubKey: <m> <pubKey> [<pubKey> . . . ] <n> OP_CHECKMULTISIG
ScriptSig: 0 [<sig > . . . <sign>]
```

  multisig is usually part of the P2SH redeem script, mentioned in the previous bullet point

- **Pay to Pubkey:** This script is a very simple script that is commonly used in **coinbase transactions**. It is now obsolete and was used in an old version of bitcoin. The public key is stored within the script in this case, and the unlocking script is required to sign the transaction with the private key.

  <pubKey> OP_CHECKSIG

Types of transactions

- **Null data/OP_RETURN**: This script is used to store arbitrary data on the blockchain for a fee. The limit of the message is **40 bytes**. The output of this script is unredeemable because OP_RETURN will fail the validation in any case. ScriptSig is not required.

  OP_RETURN <data>

- A P2PKH script execution is shown in the following diagram: