

Bitcoin: A Comprehensive Introduction

NIT Trichy

Mahavir Jhavar

Department of Computer Science
Ashoka University

June 27, 2021

Bitcoin: A (Very) Brief Introduction!

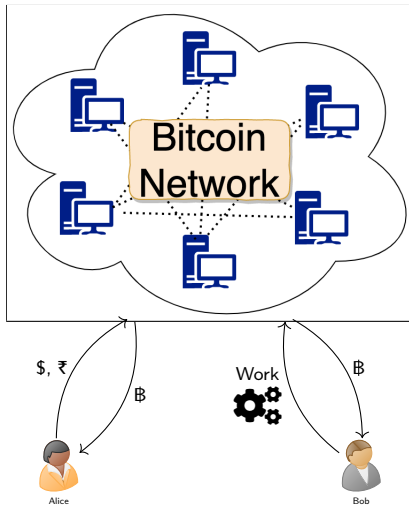
- Bitcoin is both a currency and a payment network!!
-

- How to Introduce Bitcoin?
 - ① **One way:** discuss how a traditional fiat currency and different payment networks for fiat currency work and what different problems they face. Then introduce Bitcoin as a breakthrough solution !!
 - ② **Another way:** Start discussing Bitcoin right away and present its new features as the discussion progresses!
-

- We will opt for ②!!

* <https://sites.google.com/view/brgashoka/blockchain-course?authuser=1>

How to acquire Bitcoins?



Receiving Bitcoins (₿)

● Opening bank account vs Bitcoin account!

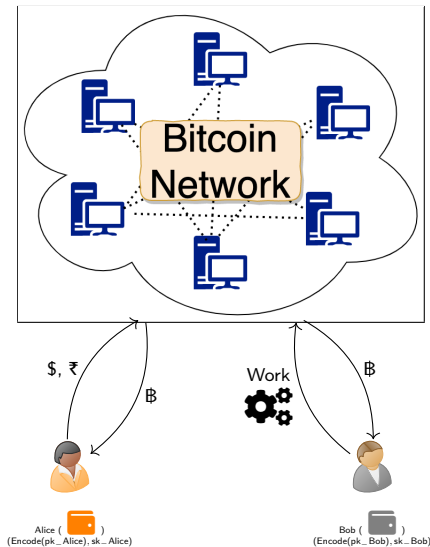
● Bank Account

- (A/C No, Secret Pins) \leftarrow Users in banking system
- A/C No is used to receive money
- Secret Pins for establishing ownership of money
- Secret Pins for approving transactions on your money

● Bitcoin Account

- $(pk, sk) \leftarrow$ Users in Bitcoin
- (A/C No, Secret Pins) \equiv (Encode(pk), sk)
- Bitcoin address = Encode(pk)
- Bitcoin address - to receive ₿
- sk - to establish ownership of money
- sk - approve transactions on your money

How to acquire Bitcoins?



► secp256k1 curve (Bitcoin) parameters

► $p : 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$

► $E : y^2 = x^3 + 7$, defined over \mathbb{Z}_p

► $G = (x, y) \in \mathcal{G}_E$ is the base point, where
 $x =$

79be667e f9dcbba c55a06295 ce870b07
029bfcdb 2dce28d9 59f2815b 16f81798

and $y =$

483ada77 26a3c465 5da4fbfc 0e1108a8
fd17b448 a6855419 9c47d08f fb10d4b8

► $\text{order}(G) = q$ is a prime, where $q =$

FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
BAAEDCE6 AF48A03B BFD25E8C D0364141

KeyGen

► Pick $a \xleftarrow{\$} \mathbb{Z}_q$. Compute $H = aG$

► Set $\begin{cases} \text{pk} = (E, p, q, G, H) \\ \text{sk} = (a, \text{pk}) \end{cases}$

Sign($\text{msg} = m, \text{sk}$)

1. Compute $h = \text{SHA256}(\text{SHA256}(m))$

2. Pick $r \xleftarrow{\$} \mathbb{Z}_q$. Compute rG .

3. Let $rG = (u, v)$, where $u, v \in \mathbb{Z}_p$

4. $c_1 = u \bmod q$ (If $c_1 = 0$, goto (2))

5. $c_2 = r^{-1}(h + ac_1) \bmod q$ (If $c_2 = 0$, goto (2))

6. Output signature $\sigma = (c_1, c_2)$

Verify($m, \sigma = (c_1, c_2), \text{pk}$)

► Compute $h = \text{SHA256}(\text{SHA256}(m))$

► Compute $e_1 = hc_2^{-1} \bmod q$

► Compute $e_2 = c_1c_2^{-1} \bmod q$

► Compute $J = e_1G + e_2H$

► Let $J = (u, v)$

► Output **valid** $\iff u \bmod q = c_1$

ECDSA pk Serialization

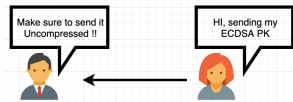
- Let $H = (z_1, z_2)$ be a ECDSA pk

- **pk-UnCompressed:**

- `0x04||hexify(z1)||hexify(z2)`
- Example: 04B20F34AE . . . 42A3C4FE

- **pk-Compressed**

- Key idea: Given z_1 , it is easy to compute z_2 .
- $z_2 = \pm \sqrt{(z_1^3 + az_1 + b)} \pmod{p}$.
- Therefore, pk = $H = (z_1, z_2)$ is completely recoverable just from the knowledge of z_1 and the sign of z_2 !
- This is achieved as follows:
- Note that, the square roots $\pm \sqrt{(z_1^3 + az_1 + b)}$ are of different parity- one is odd and the other is even.
- If z_2 is a even number, then pk-Compressed = `0x02||hexify(z1)`
- If z_2 is a odd number, then pk-Compressed = `0x03||hexify(z1)`



Base58 Encoding

Input: Byte string $\mathbf{b} = \bar{b}_n \bar{b}_{n-1} \dots \bar{b}_0$, $\bar{b}_i \in \{0, 1\}^8$

Output: Base58 encoding of \mathbf{b}

1. Encode each leading zero byte (if any) as a 1.
2. Let \bar{b}_k be the first byte such that $\bar{b}_k \neq \bar{0}$
3. Compute $S = (\bar{b}_k)_{10} 256^k + \dots + (\bar{b}_1)_{10} 256 + (\bar{b}_0)_{10}$, where $(\bar{b}_i)_{10}$ represents the decimal conversion.
4. Convert S to a base 58 representation $c_\ell \dots c_1 c_0$, i.e., $S = c_\ell 58^\ell + \dots + c_1 58 + c_0$.
5. Map each integer c_i in $0 \leq c_i \leq 57$ to the corresponding Base58 character (see table) and append the resulting string to the $n - k$ 1's (from Step 1.).

● **Input:** 0x000000261913

- Leading zero bytes 0x00 00 00 26 19 13
- $(S)_{256} = 38 \ 25 \ 19$.
- Thus $S = 2496787$
- $(S)_{58} = 12 \ 46 \ 12 \ 3$.

Base58 Integer to Character Mapping

Int	Ch	Int	Ch	Int	Ch	Int	Ch	Int	Ch	Int	Ch
0	1	9	A	18	K	27	U	36	d	44	m
1	2	10	B	19	L	28	V	37	e	45	n
2	3	11	C	20	M	29	W	38	f	46	o
3	4	12	D	21	N	30	X	39	g	47	p
4	5	13	E	22	P	31	Y	40	h	48	q
5	6	14	F	23	Q	32	Z	41	i	49	r
6	7	15	G	24	R	33	a	42	j	50	s
7	8	16	H	25	S	34	b	43	k	51	t
8	9	17	J	26	T	35	c	44	m	52	u

- the number "0", the uppercase letter "O", the upper case letter "I" (of i) and the lower case letter "l" (ell) are excluded to avoid confusion.

● **Input:** 0x000000261913

- Leading zero bytes 0x00 00 00 26 19 13
- $(S)_{256} = 38\ 25\ 19$.
- Thus $S = 2496787$
- $(S)_{58} = 12\ 46\ 12\ 3$.

● **Output:** 111DoD4

Bitcoin Address: P2PKH Address Type

Input: Public Key pk

Output: P2PKH Address

1. Compute $H1 = \text{SHA256}(pk)$
2. Compute $H2 = \text{RIPEMD160}(H1)$
3. Set $H3 = 0x00 \parallel H2$
4. Compute $H4 = \text{SHA256}(\text{SHA256}(H3))$
5. Set $H5 = H3 \parallel H4[1 : 4] = A \parallel H2 \parallel H4[1 : 4]$, where $H4[1 : 4]$ denotes the first 4 bytes of $H4$.
6. Output P2PKH address as

$\text{Base58Encoding}(H5)$

- **Note:** Easy to get $\text{RIPEMD160}(\text{SHA256}(pk)) \leftarrow$ P2PKH Address
- Denote: $\text{HASH160}(pk) = \text{RIPEMD160}(\text{SHA256}(pk))$

Serialisation: sk, pk, P2PKH Addresses

- Generate sk

sk Format	Prefix
hex	
WIF	5
WIF-Comp.	K/L

- $pk \leftarrow sk$

pk Format	Prefix
Un-Comp.	04
Comp.	02/03

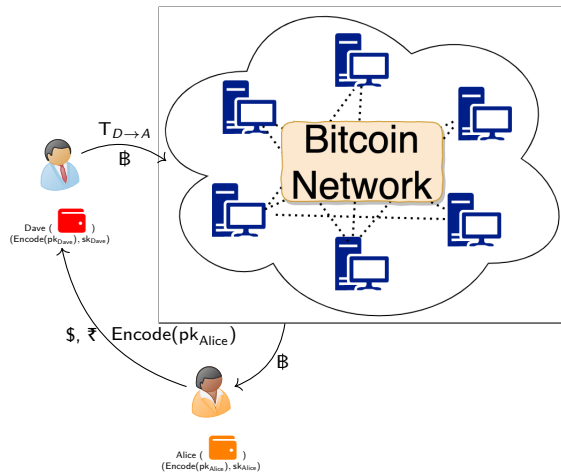
- Bitcoin Address $\leftarrow pk$

Address Format	Prefix
P2PK	
P2PKH (Base58)	mainnet(1), testnet (m/n)
P2PKH (Bech32)	mainnet(bc1), testnet (tb)
P2SH	

► Few examples:

- 1Dw1AaNiNVuCBL2nhJdE6tqRsWfg9W8GH8
- bc1qzttylpu7m8gsaph3aj4fwquhjlgy34lmm2ysgm
- 34De3LyvhoJo5VvnRaMoyhQrutN6EGz5Ua
- $\text{length}(\text{Bitcoin Address}) \leq 34$

Who gives you Bitcoins?

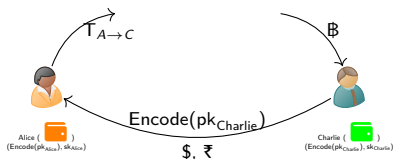
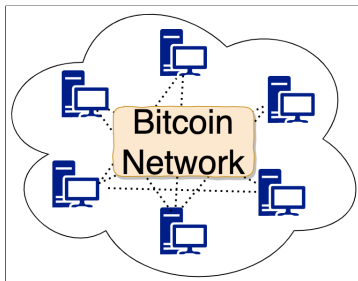


Public Ledger
⋮
$T_{D \rightarrow A}$
⋮

txn	Input		Unspent txn Output (UTXO)	
	txid	scriptSig	Amount	scriptPubkey (LockScript)
$T_{D \rightarrow A}$			$x \text{ ₿}$	$\left\{ \begin{array}{l} \text{Output pre-image of } \text{HASH}(\text{pk}_{\text{Alice}}) \\ \text{Provide a sample signature using } \text{sk}_{\text{Alice}} \end{array} \right.$

Who gives you Bitcoins?

Dave ()
 $(\text{Encode}(\text{pk}_{\text{Dave}}, \text{sk}_{\text{Dave}}))$



Public Ledger

...

$T_{D \rightarrow A}$

...

$T_{A \rightarrow C}$

...

txn	Input		Unspent txn Output (UTXO)	
	txid	scriptSig	Amount	scriptPubkey (LockScript)
$T_{A \rightarrow C}$	Ref. $T_{D \rightarrow A}$	Output $(\text{pk}_{\text{Alice}}, \sigma = \text{Sign}(T_{A \rightarrow C}, \text{sk}_{\text{Alice}}))$	$\times \mathbb{B}$	$\left\{ \begin{array}{l} \text{Output pre-image of HASH}(\text{pk}_{\text{Charlie}}) \\ \text{Provide a sample signature using sk}_{\text{Charlie}} \end{array} \right.$

Receiving Bitcoins (₿): Transactions

- Let $A(pk_A, sk_A)$ be a Bitcoin user.
 - Let $D(pk_D, sk_D)$ be another Bitcoin user who wish to transfer Bitcoins to A .
 - For this D must acquire $P2PKH_{pk_A}$
 - In the following we give two transactions (partial views) - the first one sending money to A ; the second one describes how A can spend them subsequently (transferring to C)

● $T_{D \rightarrow A}$

txn	Input		Output	
	txid	scriptSig	UTXO	scriptPubkey
$T_{D \rightarrow A}$			$\times \text{₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 (HASH160(pk}_A\text{))} \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right\}$

● $T_{A \rightarrow C}$

txn	Input		Output	
	txid	scriptSig	UTXO	scriptPubkey
$T_{A \rightarrow C}$	$H(T_{D \rightarrow A})$	$\langle \text{Sign}_{sk_A}(T_{A \rightarrow C}) \rangle \langle pk_A \rangle$	$\times \text{₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 (HASH160(pk}_C\text{))} \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right\}$

Transaction Validation: $T_{A \rightarrow C}$

txn	Input		Output	
	txid	scriptSig	UTXO	scriptPubkey
$T_{D \rightarrow A}$			$\times \mathbb{B}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 } \langle \text{HASH160}(pk_A) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right.$

txn	Input		Output	
	txid	scriptSig	UTXO	scriptPubkey
$T_{A \rightarrow C}$	$H(T_{D \rightarrow A})$	$\langle \text{Sign}_{sk_A}(T_{A \rightarrow C}) \rangle \langle pk_A \rangle$	$\times \mathbb{B}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 } \langle \text{HASH160}(pk_C) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right.$

● scriptPubkey

OP_DUP OP_HASH160 <Public Key Hash> OP_EQUAL OP_CHECKSIG

● Corresponding scriptSig

<Signature> <Public Key>

-
- scriptPubkey $\equiv \text{LS}_{\text{UTXO}}$
 - scriptSig $\equiv \text{ULS}_{\text{UTXO}}$

Executing Unlocking, Locking Scripts in Sequence

► $U L S_{U T X O} || L S_{U T X O}$

► \Downarrow
► $\langle \text{Signature} \rangle \langle \text{Public Key} \rangle \text{op_dup op_hash160} \langle \text{Public Key Hash} \rangle \text{op_equal op_checksig}$



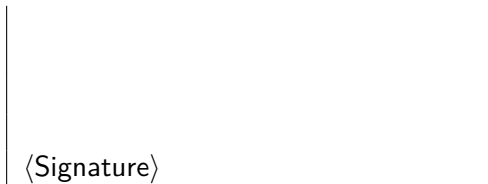
Stack

► \Downarrow points to the next argument to be executed

Executing Unlocking, Locking Scripts in Sequence

► $ULS_{UTXO} || LS_{UTXO}$

► $\langle \text{Signature} \rangle \langle \text{Public Key} \rangle$ \Downarrow `op_dup op_hash160 $\langle \text{Public Key Hash} \rangle$ op_equal op_checksig`



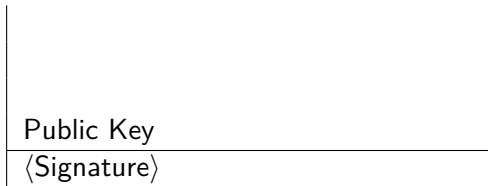
Stack

► \Downarrow points to the next argument to be executed

Executing Unlocking, Locking Scripts in Sequence

► $ULS_{UTXO} || LS_{UTXO}$

► $\langle \text{Signature} \rangle \langle \text{Public Key} \rangle \downarrow \text{op_dup op_hash160 } \langle \text{Public Key Hash} \rangle \text{ op_equal op_checksig}$



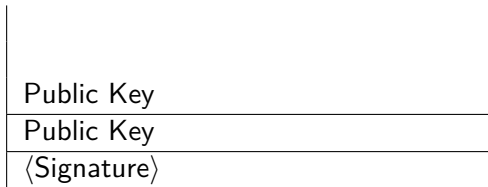
Stack

► \downarrow points to the next argument to be executed

Executing Unlocking, Locking Scripts in Sequence

► $ULS_{UTXO} || LS_{UTXO}$

► $\langle \text{Signature} \rangle \langle \text{Public Key} \rangle \text{op_dup op_hash160} \langle \text{Public Key Hash} \rangle \text{op_equal op_checksig}$



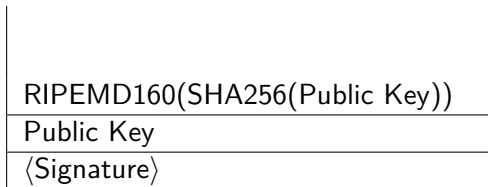
Stack

► \Downarrow points to the next argument to be executed

Executing Unlocking, Locking Scripts in Sequence

► $ULS_{UTXO} || LS_{UTXO}$

► $\langle \text{Signature} \rangle \langle \text{Public Key} \rangle \text{op_dup op_hash160} \langle \text{Public Key Hash} \rangle \text{op_equal op_checksig}$



Stack

► \Downarrow points to the next argument to be executed

Executing Unlocking, Locking Scripts in Sequence

► $ULS_{UTXO} || LS_{UTXO}$

► $\langle \text{Signature} \rangle \langle \text{Public Key} \rangle \text{op_dup op_hash160} \langle \text{Public Key Hash} \rangle \text{op_equal op_checksig}$ \Downarrow

$\langle \text{Public Key Hash} \rangle$
$\text{RIPEMD160}(\text{SHA256}(\text{Public Key}))$
Public Key
$\langle \text{Signature} \rangle$

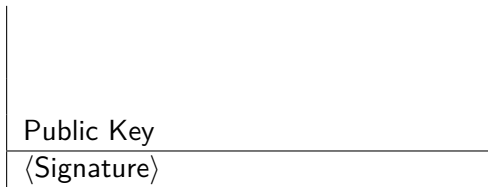
Stack

► \Downarrow points to the next argument to be executed

Executing Unlocking, Locking Scripts in Sequence

► $ULS_{UTXO} || LS_{UTXO}$

► $\langle \text{Signature} \rangle \langle \text{Public Key} \rangle \text{op_dup op_hash160 } \langle \text{Public Key Hash} \rangle \text{op_equal op_checksig}$



Stack

► \Downarrow points to the next argument to be executed

Executing Unlocking, Locking Scripts in Sequence

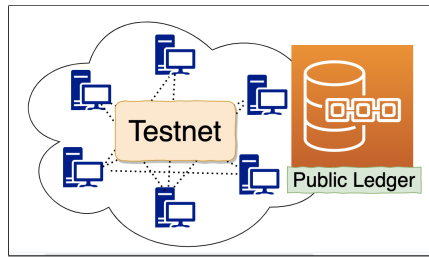
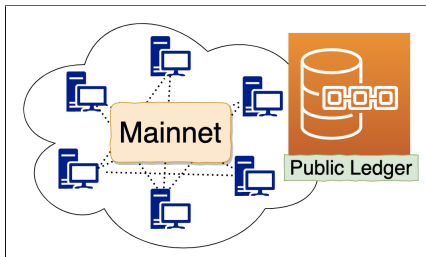
- ▶ $ULS_{UTXO} || LS_{UTXO}$
- ▶ $\langle \text{Signature} \rangle \langle \text{Public Key} \rangle \text{ op_dup op_hash160 } \langle \text{Public Key Hash} \rangle \text{ op_equal op_checksig}$



Stack

- ▶ \Downarrow points to the next argument to be executed

Bitcoin: Mainnet Vs Testnet



- Search engine for Bitcoin networks (both mainnet and testnet): Bitcoin Explorers
 - <https://www.blockchain.com/explorer>
 - <https://blockchair.com/bitcoin>
 - <https://live.blockcypher.com/btc/>

Transaction: Multiple Inputs, Multiple Outputs

Multiple Inputs, Single Output

txn	Input		Unspent txn Output (UTXO)	
	txid	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{B \rightarrow C}$	$H(T_{A_1 \rightarrow B})$ $H(T_{A_2 \rightarrow B})$ $H(T_{A_3 \rightarrow B})$	$\langle \text{Sign}_{sk_B}(T_{B \rightarrow C}) \rangle \langle pk_B \rangle$ $\langle \text{Sign}_{sk_B}(T_{B \rightarrow C}) \rangle \langle pk_B \rangle$ $\langle \text{Sign}_{sk_B}(T_{B \rightarrow C}) \rangle \langle pk_B \rangle$	$x \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 (HASH160(pk}_C\text{))} \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right\}$

Multiple Inputs, Multiple Outputs

txn	Input		Unspent txn Output (UTXO)	
	txid	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{B \rightarrow \{C_1, C_2\}}$	$H(T_{A_1 \rightarrow B})$	$\langle \text{Sign}_{sk_B}(T_{B \rightarrow \{C_1, C_2\}}) \rangle \langle pk_B \rangle$	$x_1 \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 (HASH160(pk}_{C_1}\text{))} \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right\}$
	$H(T_{A_2 \rightarrow B})$	$\langle \text{Sign}_{sk_B}(T_{B \rightarrow \{C_1, C_2\}}) \rangle \langle pk_B \rangle$		
	$H(T_{A_3 \rightarrow B})$	$\langle \text{Sign}_{sk_B}(T_{B \rightarrow \{C_1, C_2\}}) \rangle \langle pk_B \rangle$	$x_2 \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 (HASH160(pk}_{C_2}\text{))} \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right\}$

Transaction: Spending Selective Outputs

①

txn	Input		Unspent txn Output (UTXO)	
	txid	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{B \rightarrow \{C_1, C_2\}}$	$H(T_{A_1 \rightarrow B})$	$\langle \text{Sign}_{sk_B}(T_{B \rightarrow \{C_1, C_2\}}) \rangle \langle pk_B \rangle$	$x_1 \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 } \langle \text{HASH160}(pk_{C_1}) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right\}$
	$H(T_{A_2 \rightarrow B})$	$\langle \text{Sign}_{sk_B}(T_{B \rightarrow \{C_1, C_2\}}) \rangle \langle pk_B \rangle$		
	$H(T_{A_3 \rightarrow B})$	$\langle \text{Sign}_{sk_B}(T_{B \rightarrow \{C_1, C_2\}}) \rangle \langle pk_B \rangle$	$x_2 \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 } \langle \text{HASH160}(pk_{C_2}) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right\}$

②

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1 \rightarrow D}$	$H(T_{B \rightarrow \{C_1, C_2\}})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1 \rightarrow D}) \rangle \langle pk_{C_1} \rangle$	$x_1 \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 } \langle \text{HASH160}(pk_D) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right\}$

Transaction: Consuming UTXO Amounts in its Entirety

- Suppose a user C_1 wants to pay $x \text{ ₿}$ to another user D , but the UTXO that C_1 owns is worth $x_1 (> x)$ ₿ given as follows

txn	Input		Unspent txn Output (UTXO)	
	txid	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{B \rightarrow \{C_1, C_2\}}$	$H(T_{A_1 \rightarrow B})$	$\langle \text{Sign}_{sk_B}(T_{B \rightarrow \{C_1, C_2\}}) \rangle \langle pk_B \rangle$	$x_1 \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 } \langle \text{HASH160}(pk_{C_1}) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right.$
	$H(T_{A_2 \rightarrow B})$	$\langle \text{Sign}_{sk_B}(T_{B \rightarrow \{C_1, C_2\}}) \rangle \langle pk_B \rangle$		
	$H(T_{A_3 \rightarrow B})$	$\langle \text{Sign}_{sk_B}(T_{B \rightarrow \{C_1, C_2\}}) \rangle \langle pk_B \rangle$	$x_2 \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 } \langle \text{HASH160}(pk_{C_2}) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right.$

- In Bitcoin, the entirety of a transaction output must be consumed by another transaction, or none of it
 - So, in preparing $T_{C_1 \rightarrow D}$, C_1 needs to create a new output where $(x_1 - x) \text{ ₿}$ are sent back to C_1 by locking it to pk_{C_1} .
 - It could be a different address from the one that owned the $x_1 \text{ ₿}$, but it would have to be owned by C_1

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1 \rightarrow D}$	$H(T_{B \rightarrow \{C_1, C_2\}})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1 \rightarrow D}) \rangle \langle pk_{C_1} \rangle$	$x \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 } \langle \text{HASH160}(pk_D) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right.$
				$(x_1 - x) \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 } \langle \text{HASH160}(pk_{C_1}) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right.$

Transaction Fee

Multiple Inputs, Single Output



txn	Input		Unspent txn Output (UTXO)	
	txid	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{B \rightarrow C}$	$H(T_{A_1 \rightarrow B})$	$\langle \text{Sign}_{sk_B}(T_{B \rightarrow C}) \rangle \langle pk_B \rangle$	$x \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 } \langle \text{HASH160}(pk_C) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right.$
	$H(T_{A_2 \rightarrow B})$	$\langle \text{Sign}_{sk_B}(T_{B \rightarrow C}) \rangle \langle pk_B \rangle$		
	$H(T_{A_3 \rightarrow B})$	$\langle \text{Sign}_{sk_B}(T_{B \rightarrow C}) \rangle \langle pk_B \rangle$		

Multiple Inputs, Multiple Outputs

txn	Input		Unspent txn Output (UTXO)	
	txid	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{B \rightarrow \{C_1, C_2\}}$	$H(T_{A_1 \rightarrow B})$	$\langle \text{Sign}_{sk_B}(T_{B \rightarrow \{C_1, C_2\}}) \rangle \langle pk_B \rangle$	$x_1 \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 } \langle \text{HASH160}(pk_{C_1}) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right.$
	$H(T_{A_2 \rightarrow B})$	$\langle \text{Sign}_{sk_B}(T_{B \rightarrow \{C_1, C_2\}}) \rangle \langle pk_B \rangle$	$x_2 \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 } \langle \text{HASH160}(pk_{C_2}) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right.$
	$H(T_{A_3 \rightarrow B})$	$\langle \text{Sign}_{sk_B}(T_{B \rightarrow \{C_1, C_2\}}) \rangle \langle pk_B \rangle$		

$$x = x_1 + x_2 + \text{Fee}$$

A Real Transaction

Summary ⓘ				USD	BTC
Hash	f7c6c4873bbe4b93d50680feb3a28b076db241d2e90003f9c54... 			2021-02-15 21:43	
	14zKiUeq2MNow7s1GscXhta8VNmckdsc6V	0.07181656 BTC 	➡	1GcwgaxRzq8NKosQSD3cjB8Mkub82f224r 1Hu6qGMfUfqCXXBrnCSH7FhtExqs8F86FY	0.06099847 BTC  0.01055900 BTC 
Fee	0.00025909 BTC (114.642 sat/B - 28.660 sat/WU - 226 bytes)			0.07155747 BTC	

► $0.06099847 + 0.01055900 + 0.00025909 = 0.07181656$

Inputs ⓘ



HEX

ASM

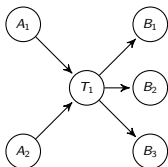
Index	0	Details	Output
Address	14zKiUeq2MNow7s1GscXhta8VNmckdsc6V ⓘ	Value	0.07181656 BTC
Pkscript	OP_DUP OP_HASH160 2bbfb5b337d7f76ca7555d05e9170586ce843c7e OP_EQUALVERIFY OP_CHECKSIG		
Sigscript	3045022100be97f47f15411de43a4b50acf14deb6a9220a0afaa9aa039ddce3a7276c076cc02205199a1363c57eccf173072e36488883d846454e3462bf79d28d065983a30cba001 0359038bd7153b7f9c9ec5ede4f40ec20d3b17288469ff01b83c1f8aaae7a7ad1d		

A Real Transaction

Outputs 1

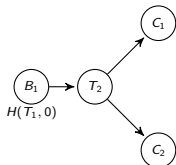
Index	0	Details	Unspent
Address	1GcwgaxRzq8NKosQSD3cjB8Mkub82f224r 	Value	0.06099847 BTC
Pkscript	OP_DUP OP_HASH160 ab5617926b20242253bf36fdf9a83b1fa49a6959 OP_EQUALVERIFY OP_CHECKSIG		
Index	1	Details	Unspent
Address	1Hu6qGMfUfqCXXBrnCSH7FhtExqs8F86FY 	Value	0.01055900 BTC
Pkscript	OP_DUP OP_HASH160 b95c82d01cbc34bf778502fefbcdcf2787923708e OP_EQUALVERIFY OP_CHECKSIG		

Bitcoin Ledger and Worldstate

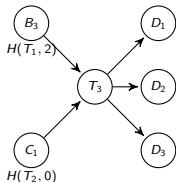


Worldstate

txid	S.N.	Status
$H(T_1)$	0	Unspent
$H(T_1)$	1	Unspent
$H(T_1)$	2	Unspent

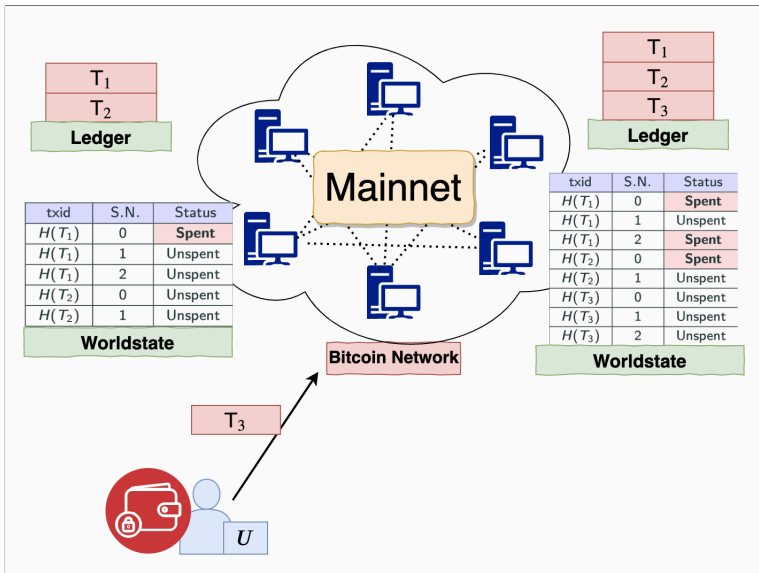


txid	S.N.	Status
$H(T_1)$	0	Spent
$H(T_1)$	1	Unspent
$H(T_1)$	2	Unspent
$H(T_2)$	0	Unspent
$H(T_2)$	1	Unspent

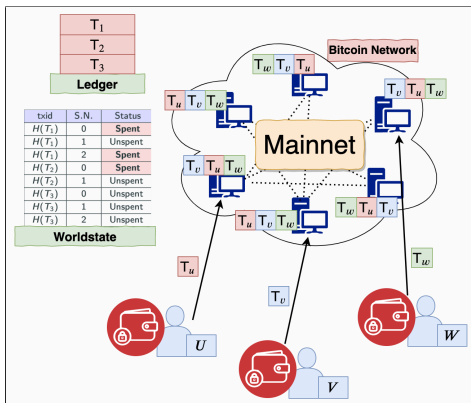


txid	S.N.	Status
$H(T_1)$	0	Spent
$H(T_1)$	1	Unspent
$H(T_1)$	2	Spent
$H(T_2)$	0	Spent
$H(T_2)$	1	Unspent
$H(T_3)$	0	Unspent
$H(T_3)$	1	Unspent
$H(T_3)$	2	Unspent

Bitcoin Ledger and Worldstate



Working of the Bitcoin Network



- When a user A wants to pay user B , what actually does is broadcast a transaction to the Bitcoin nodes that comprise the peer-to-peer network.
- Given that a variety of users are broadcasting these transactions to the network, **the nodes must agree on exactly which transaction is valid, the order in which these transactions happened.**
- This will result in a single, global ledger system

- The nodes in the Bitcoin network must reach a consensus on all the transactions proposed and the order in which these transactions happened !!
- How difficult is this to achieve??

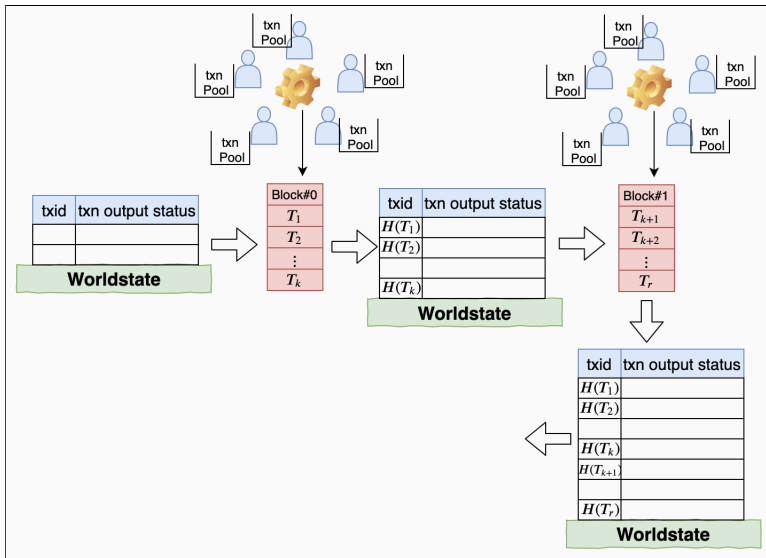
Bitcoin's Distributed Consensus Protocol

- The Bitcoin's distributed consensus protocol helps the network to reach a consensus on a collection (block) of transactions at one go !!
 - That is, it runs consensus on a block-by-block basis.

- Bitcoin's Distributed Consensus Protocol (A beginning)

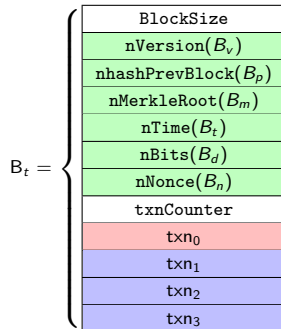
- **Assumption:** Each node receives and maintains all unapproved transactions broadcasted into the network
- ▶ **Step1** The starting of the Consensus Protocol.
 - Each node collects a few transaction from its pool of unapproved transactions and begins to prepare a "**block by using the selected transactions.**"
 - Preparing this block is not an easy task !!
 - Whoever prepare it firsts will basically decided the next set of transactions to be added to the ledger!! Basically, all those transactions that were used by the winning node to prepare this block.
 - All other nodes must agree to this as long as the transactions included in the winning block are all valid !!
 - If some transaction somehow didn't make it into this particular block, it could just wait and get into the next block.

The Working of the Network



How to prepare a block? - Mining

- ▶ Assume, so far $t - 1$ blocks were accepted by the network
 - Now is the turn for a t -th block to be prepared and accepted by the network.
- ▶ Suppose, a node wants to prepare the t th block using three txns: txn_1 , txn_2 and txn_3 .
- The node also puts a special transactions txn_0 into the block!!

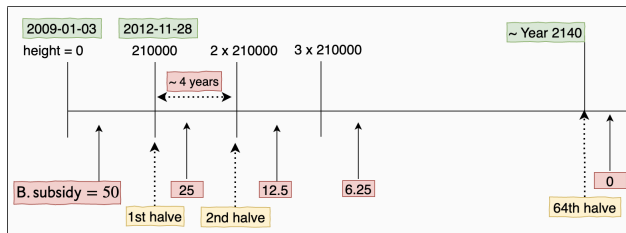


txn₀ - The Coinbase Transaction

- The first transaction in any block B is a special transaction, called a **coinbase transaction**.
 - A coinbase transaction is constructed by a miner who is preparing the block.
 - The transaction contains miner's reward for the mining effort.
 - A block reward is computed as follows:
- $\boxed{B.\text{reward}} = \boxed{B.\text{subsidy}} + \boxed{\sum_{\text{txn} \in B} \text{txn}.\text{fee}}$, where B.subsidy is computed as follows:

$$B.\text{subsidy} = \frac{50}{2^\ell} \text{ ₿}, \text{ where } \ell = \left\lfloor \frac{\text{height}(B)}{210000} \right\rfloor$$

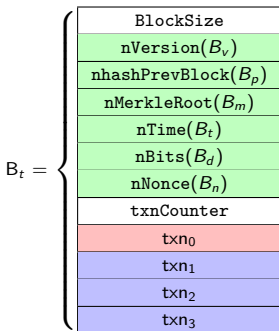
- The block B₆₇₃₂₂₅ is mined on 2021-03-05 12:26. Therefore, $\ell = \left\lfloor \frac{673225}{210000} \right\rfloor = \lfloor 3.2 \rfloor = 3$.
- This means, $B_{673225}.\text{subsidy} = \frac{50}{2^3} = \frac{50}{8} = 6.25 \text{ ₿}$



How to prepare a block? - Mining

- ▶ Assume, so far $t - 1$ blocks were accepted by the network
 - This is race for t th block !!
- ▶ Suppose, a node wants to prepare the t th block using three txns: txn_1, txn_2 and txn_3 .

- The block structure



- ▶ **BlockSize**: number of bytes following up to end of block.

- Uses 4 bytes to represent Blocksize

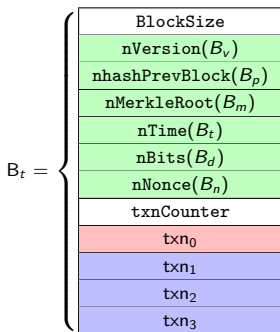
- ▶ **Block Header**: $B_t^h = [B_v || B_p || B_m || B_t || B_d || B_n]$

- nVersion(B_v): 4-byte field, specifies the version of the block.

- nhashPrevBlock(B_p): $H(B_{t-1}^h)$

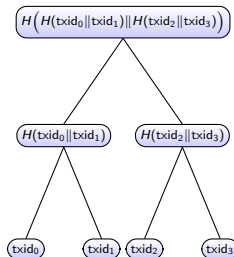
Notation: $H(\cdot) = \text{SHA256}(\text{SHA256}(\cdot))$

Block Preparation: Mining

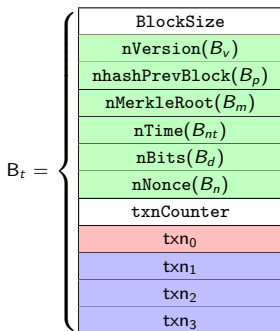


● $nMerkleRoot = H(H(txid_0 || txid_1) || H(txid_2 || txid_3))$

- And it is computed as follows



Block Preparation: Mining



► **BlockSize**: number of bytes following up to end of block.

- Uses 4 bytes to represent Blocksize

► **Block Header**: $B_t^h = [B_v || B_p || B_m || B_t || B_d || B_n]$

● nVersion(B_v): 4-byte field, specifies the version of the block.

● nhashPrevBlock(B_p): $H(B_{t-1}^h)$

● nMerkleRoot (B_m):

● nTime (B_{nt}): is a timestamp in Unix time format to record the time of candidate block creation.

● nBits (B_d), and nNonce (B_n) ??

Block Preparation: Mining

$B_t =$ {

BlockSize
nVersion(B_v)
nhashPrevBlock(B_p)
nMerkleRoot(B_m)
nTime(B_{nt})
nBits(B_d)
nNonce(B_n)
txnCounter
txn ₀
txn ₁
txn ₂
txn ₃

- In order to prepare a valid block, a miner must solve a difficult computational problem.
 - Every miner gets the same problem
 - A miner puts the encoded representation of the problem into the `nBits` field in the header.
 - After solving the problem, a miner puts the solution into the `nNonce` field in the header.
 - Consequently, the block is now ready (mined successfully) as a valid block
- _____ We now discuss,
- 1 The problem description
 - 2 The encoded representation of the problem
 - 3 How difficult is the problem?
 - 4 How is the problem constructed/determined?

Block Preparation: Mining

① The problem description

-
- **Input:** $T \in \{0, 1, \dots, 2^{256} - 1\}$
 - **Output:** Prepare B_t^h such that $H(B_t^h) < T$
-

- The values in the following **fields** in B_t^h are not in a miner's control:

$$B_t^h = [B_v \| B_p \| B_m \| B_{nt} \| B_d \| B_n]$$

- $B_m = \text{MerkleRoot}(\text{txn}_0[[r]], \text{tx}_1, \text{tx}_2, \text{tx}_3)$, where $r \in \{0, 1\}^{\leq 768}$ and $\text{txn}_0[[r]]$ is a function of r
 - And, $B_n \in \{0, 1\}^{32}$
-

- Therefore, **Output:** Find B_m and B_n such that $H(B_t^h) < T$
 - Equivalently, the problem is described as follows
-

$$P_T = \begin{cases} \text{Input: } X \in \{0, 1\}^{44 \times 8} \text{ and } T \in \{0, 1, \dots, 2^{256} - 1\} \\ \text{Output: Find } Y \in \{0, 1\}^{36 \times 8} \text{ such that } H(X \| Y) < T \end{cases}$$

where $X = [B_v \| B_p \| B_{nt} \| B_d]$ and $Y = [B_m \| B_n]$

Block Preparation: Mining

- 2 The encoded representation of the problem.

- **Input** $T \in \{0, 1, \dots, 2^{256} - 1\}$
- **Output** $T_{e||c} \in \{0, 1\}^{32}$ where

$$T_{e||c} = 0xd_1 d_2 \dots d_7 d_8$$

and d_i 's are hex numbers.

- T can be recovered from $T_{e||c}$ as follows

$$T = c \times (256)^{8 \times (e-3)}$$

where $e = (0xd_1 d_2)_{10}$ and $c = (0xd_3 \dots d_8)_{10}$ are decimal numbers.

An example

- The first Block, i.e., B_0 , was mined by "**Satoshi**" on 2009-01-03 23:45 [See - Blockchain.com explorer]
- The nBits value given in its header = 486604799
- Thus, $T_{e||c} = \text{hex}(486604799) = 0x1d00ffff$
- This means,

$$T = c \times (256)^{8 \times (e-3)}$$

$$= (0x00ffff)_{10} \times (256)^{8 \times ((0x1d)_{10} - 3)}$$

$$= 65535 \times (256)^{8 \times (29-3)}$$

$$= 26959535291011309493156476344723991336010898738574164086137773096960$$

Block Preparation: Mining

③ How difficult is the problem?

- Note that, every instance P_T of this problem comprises of T -many pre-image finding problems with respect to the underlying hash function $H(\cdot)$
- Recall: pre-image finding problem

$$PF_b = \begin{cases} \text{Input: } H : \{0, 1\}^* \rightarrow \{0, 1\}^{256} \text{ and } b \in \{0, 1\}^{256} \\ \text{Output: } a \in \{0, 1\}^* \text{ such that } H(a) = b \end{cases}$$

- Therefore, for a $T \in \{0, 1, \dots, 2^{256} - 1\} = \{0, 1\}^{256}$

$$P_T \equiv \{PF_b \mid 0 \leq b \leq T\}$$

- **Fact** For any $b \in \{0, 1, \dots, 2^{256} - 1\}$ we have

$$\mathbb{P}[H(a) = b \mid a \text{ is picked randomly}] = \frac{1}{2^{256}}$$

- The above **Fact** implies that the best algorithm to solve PF_b is the following:

While $H(a) = b$
 Pick a randomly
 Compute $H(a)$
Output a

- Therefore, the best algorithm to solve P_T is

While $H(a) \in \{0, 1, \dots, T - 1\}$
 Pick a randomly
 Compute $H(a)$
Output a

Block Preparation: Mining

- ③ How difficult is the problem?

$$P_T = \begin{cases} \text{Input: } X \in \{0, 1\}^{44 \times 8} \text{ and } T \in \{0, 1, \dots, 2^{256} - 1\} \\ \text{Output: Find } Y \in \{0, 1\}^{36 \times 8} \text{ such that } H(X \| Y) < T \end{cases}$$

- The best algorithm to solve P_T is:

While $H(X \| Y) \in \{0, 1, \dots, T - 1\}$
 Pick Y randomly
 Compute $H(X \| Y)$
Output Y

- A hash is computed during each loop of the algorithm.
- Clearly, based on the **Fact** in the previous slide, the expected number of loops required before the algorithm stops is

$$\approx \frac{2^{256}}{T}$$

Block Preparation: Mining

An Example

- The Block at height 670000, i.e., B_{670000} was mined on 2021-02-10 18:39 [See Blockchain.com explorer]
- The nBits value given in its header = 386736569
- Thus, $T_{e||c} = \text{hex}(386736569) = 0x170d21b9$
- This means,

$$\begin{aligned}T &= c \times (256)^{8 \times (e-3)} \\&= (0x0d21b9)_{10} \times (256)^{8 \times ((0x17)_{10} - 3)} \\&= 860601 \times (256)^{8 \times (23-3)} \\&= 1257769770588612382309009370720465882998915202417688576\end{aligned}$$

- As $\log_2(T) \approx 180$, i.e., $T \approx 2^{180}$, the expected number of hash computations required before one finds a solution to P_T is

$$\approx \frac{2^{256}}{T} \approx \frac{2^{256}}{2^{180}} \approx 2^{76}$$

Block Preparation: Mining

- **Fact** A characterisation of $H(X\|Y)$ when $H(X\|Y) < T$ and $T \in \{0, 1\}^{256}$
 - If $\lceil \log_2(T) \rceil = k$, i.e., $T \approx 2^k$, then the binary representation of any number which is $< T$ must have at least $256 - k$ many leading 0's
 - That is, for the correct solution Y , the binary representation of $H(X\|Y)$ must have $256 - k$ many leading 0's

-
- The above observation implies that, the problem P_T can also be represented as follows:

$$P_{n\text{-bits}} = \begin{cases} \text{Input: } X \in \{0, 1\}^{44 \times 8} \text{ and } n \in \{1, \dots, 256\} \\ \text{Output: Find } Y \in \{0, 1\}^{36 \times 8} \text{ such that the binary representation of} \\ H(X\|Y) \text{ must have } n \text{ leading 0's} \end{cases}$$

- Clearly, for a $n \in \{1, \dots, 256\}$, if you set $k = 256 - n$, then

$$P_{n\text{-bits}} \equiv P_T, \text{ where } T = 2^k$$

- Indeed, binary representation of $H(X\|Y)$ having n many leading 0's implies

$$\begin{aligned} H(X\|Y) &< 2^{256-n} \\ \implies H(X\|Y) &< 2^k \\ \implies H(X\|Y) &< T \end{aligned}$$

txn Structure

Field Type	Size (bytes)
nVersion	4
Number of Inputs N	1/3/5/9
Input ₀	
⋮	
Input _{N-1}	
Number of Outputs M	1/3/5/9
Output ₀	
⋮	
Output _{M-1}	
nLockTime	4

txn Structure

Field Type	Size
hash	32
n	4
ScriptSigLen	
ScriptSig	
nSequence	4

txn Input Structure

Field Type	Size
nValue	
ScriptPubKeyLen	
ScriptPubKey	

txn Output Structure

Timelock: nLocktime

- Timelocks are restrictions on **transactions or outputs** that only allow spending after a point in time, i.e., locking funds to a date in the future.

► nLocktime - a txn-level timelock

- A nLocktime value defines the earliest time that a txn is ready for execution by the network. It is set to "0" in most txn's to indicate immediate execution as is the following txn

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{B \rightarrow \{C_1, C_2\}}$	$H(T_{A_1 \rightarrow B})$	n_1	$\langle \text{Sign}_{sk_B}(T_{B \rightarrow \{C_1, C_2\}}) \rangle \langle pk_B \rangle$	$x_1 \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160} \langle \text{HASH160}(pk_{C_1}) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right\}$
	$H(T_{A_2 \rightarrow B})$	n_2	$\langle \text{Sign}_{sk_B}(T_{B \rightarrow \{C_1, C_2\}}) \rangle \langle pk_B \rangle$		
	$H(T_{A_3 \rightarrow B})$	n_3	$\langle \text{Sign}_{sk_B}(T_{B \rightarrow \{C_1, C_2\}}) \rangle \langle pk_B \rangle$	$x_2 \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160} \langle \text{HASH160}(pk_{C_2}) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right\}$
nLockTime = 0					

- An nLocktime value is encoded using a block-height/Unix-time.
 - It is interpreted as block-height if $n\text{LockTime} < 5 \times 10^8$
 - It is Interpreted as Unix-time if $n\text{LockTime} \geq 5 \times 10^8$
- The following txn $T_{C_1 \rightarrow D}$ with $n\text{LockTime} = 674000$ will not be picked up for execution in a valid block whose height is less than 674000.
- At the time of writing $T_{C_1 \rightarrow D}$, the current time is 2021-03-01 07:30 and the block mined by the network is of height 672638

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1 \rightarrow D}$	$H(T_{B \rightarrow \{C_1, C_2\}})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1 \rightarrow D}) \rangle \langle pk_{C_1} \rangle$	$x_1 \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160} \langle \text{HASH160}(pk_D) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right\}$
nLockTime = 674000					

Timelock: nLocktime

► What about this txn?

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1 \rightarrow D}$	$H(T_{B \rightarrow \{C_1, C_2\}})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1 \rightarrow D}) \rangle \langle pk_{C_1} \rangle$	$x_1 \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160} \langle \text{HASH160}(pk_D) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right\}$
nLockTime = 1615102200					

- As $n\text{LockTime} = 1615102200 > 5 \times 10^8$, its value must be interpreted as Unix timestamp.
- One may check that 1615102200 represents "Sun Mar 07 2021 13:00:00 GMT+0530 (India Standard Time) "
- As per the rule, $T_{C_1 \rightarrow D}$ will not be picked up for execution in a valid block unless the **median-time-past** of this block is beyond Sun Mar 07 2021 13:00:00 GMT+0530

Median-time-past The median-time-past of a block at height h is the median of $\{n\text{Time}_{h-11}^{\text{block}}, n\text{Time}_{h-10}^{\text{block}}, \dots, n\text{Time}_{h-1}^{\text{block}}\}$, where $n\text{Time}_j^{\text{block}}$ denotes the nTime of the block in the blockchain at height j .

- The use of nLocktime is equivalent to postdating a paper check.
 - Indeed, after $T_{C_1 \rightarrow D}$ is prepared by C_1 and is handed over to D , it resembles like a post dated check
 - D can relay $T_{C_1 \rightarrow D}$ to the network only at a future point of time in order to receive the money.

txn Input Structure

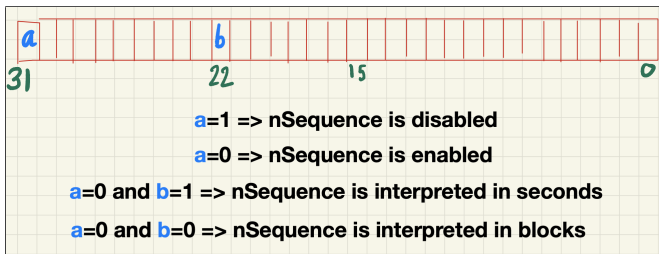
Field Type	Size
hash	32
n	4
ScriptSigLen	
ScriptSig	
nSequence	4

txn Input Structure

- `hash` Contains the transaction identifier (TXID) of an earlier txn containing the UTXO that this input tries to unlock.
- `n` Index of the UTXO in the earlier txn
- `scriptSigLen` contains VarInt encoding of the length of scriptSig
- `scriptSig` contains the unlock script - scriptSig
- `nSequence` ??

Relative Timelock: nSequence

- nLocktime implements an absolute timelock in that it specifies an absolute future time.
- nSequence implements a relative timelock in that it specifies, as a condition of spending an output, an elapsed time from the confirmation of the output in the blockchain
- Transaction inputs with nSequence enabled are interpreted as having a relative timelock. Such a transaction is only valid once the input has aged by the relative timelock amount.
- nSequence is applied to each **input** of a transaction.
- Like nLocktime, it is also possible to disable nSequence
- The nSequence value is specified in either blocks or seconds, but in a slightly different format than we saw used in nLocktime.



Relative Timelock: nSequence

- An Example

txn	Input				Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	nSequence	Amount	scriptPubkey
$T_{C_1 \rightarrow D}$	$H(T_{B \rightarrow \{C_1, C_2\}})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1 \rightarrow D}) \rangle \langle pk_{C_1} \rangle$	0x00000014	$x_1 \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 } \langle \text{HASH160}(pk_D) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right\}$
nLockTime = 1615102200						

- For 0x00400014, $a = b = 0$.
- nSequence is interpreted as 20 blocks.
- This means, $T_{C_1 \rightarrow D}$ is ready for execution when at least 20 blocks have elapsed from the time the reference txn $T_{B \rightarrow \{C_1, C_2\}}$ was accepted into a valid block by the network!

Signature Generation

- What message is chosen for signature generation??
 - Consider the signature generation for the following transaction

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1 \rightarrow D}$	$H(T_{B \rightarrow \{C_1, C_2\}})$	n	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1 \rightarrow D}) \rangle \langle pk_{C_1} \rangle$	$x_1 \text{ ₿}$	$\begin{cases} \text{OP_DUP OP_HASH160 } \langle \text{HASH160}(pk_D) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{cases}$

- That is, how to obtain $\text{Sign}_{sk_{C_1}}(T_{C_1 \rightarrow D})$??

Message for Signature Generation

● txn

nVersion
0x02
hash0
n0
scriptSigLen0
scriptSig0
nSequence0
hash1
n1
scriptSigLen1
scriptSig1
nSequence1
0x01
nValue0
scriptPubkeyLen0
scriptPubkey0
nLockTime

- scriptSig field of every txn-input contains signature.
 - For example - **scriptSig0** : $\langle \text{signature} \rangle \langle \text{pk} \rangle$
 - The $\langle \text{signature} \rangle$ is generated on a well defined message
 - The message is derived from the txn
 - Denote the message by: $m_{\text{txn},k}$ where k signals that it is for the input number k .
- The following standards dictate the selection of $m_{\text{txn},k}$

nHashType	Value
SIGHASH_ALL	0x00000001
SIGHASH_NONE	0x00000002
SIGHASH_SINGLE	0x00000003
SIGHASH_ANYONECANPAY	0x00000080

► $\text{signature} = \text{sign}_{\text{sk}}(m_{\text{txn},k}) || \text{1sb}(\text{nHashType})$

Message for Signature Generation

● txn

nVersion
0x02
hash0
n0
scriptSigLen0
scriptSig0
nSequence0
hash1
n1
scriptSigLen1
scriptSig1
nSequence1
0x01
nValue0
scriptPubkeyLen0
scriptPubkey0
nLockTime

● $m_{txn,0}$ following SIGHASH-ALL type

nVersion
0x02
hash0
n0
prevScriptPubKeyLen0
prevScriptPubKey0
nSequence0
hash1
n1
0x00
nSequence1
0x01
nValue0
scriptPubkeyLen0
scriptPubkey0
nLockTime
nHashType

nLocktime Limitations

- nLocktime has the limitation that while it makes it possible to spend some outputs in the future, it **does not make it impossible to spend them until that time.**
- Suppose, a user C_1 gives a payment transaction $T_{C_1 \rightarrow D}$ to D by setting nLocktime to 2 weeks.

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1 \rightarrow D}$	$H(T_{B \rightarrow \{C_1, C_2\}})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1 \rightarrow D}) \rangle \langle pk_{C_1} \rangle$	$x_1 \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 } \langle \text{HASH160}(pk_D) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right\}$
nLockTime = Two Weeks					

- However, there are a few issues:
- ① D cannot submit the $T_{C_1 \rightarrow D}$ to the network to redeem the money until 2 weeks have elapsed. D may submit the transaction after 2 weeks
- ② C_1 can create another transaction, **double-spending** the same inputs with a nLocktime. Thus C_1 can spend the same UTXO (that $T_{C_1 \rightarrow D}$ is going to spend after 2 weeks) before the 2 weeks have elapsed.
- D has no guarantee that C_1 won't do that.

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1 \rightarrow E}$	$H(T_{B \rightarrow \{C_1, C_2\}})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1 \rightarrow E}) \rangle \langle pk_{C_1} \rangle$	$x_1 \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 } \langle \text{HASH160}(pk_E) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right\}$
nLockTime = 0					

Check Lock Time Verify (CLTV)

- Thus, the only guarantee that nLocktime provides is that D will not be able to redeem it before 2 weeks have elapsed. There is no guarantee that D will get the money
- To achieve such a guarantee, the timelock restriction must be placed on the reference UTXO itself and be part of the locking script!!
- This is achieved by the following timelock, called "**Check Lock Time Verify (CLTV)**" _____ **The Syntax:** C_1 prepares the following transaction and immediately sends it to the network for execution.

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1 \rightarrow D}$	$H(T_{B \rightarrow \{C_1, C_2\}})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1 \rightarrow D}) \rangle \langle pk_{C_1} \rangle$	$x_1 \text{ ₿}$	$\left\{ \begin{array}{l} \langle tw \rangle \text{ OP_CHECKLOCKTIMEVERIFY OP_DROP OP_DUP} \\ \text{OP_HASH160 } \langle \text{HASH160}(pk_D) \rangle \text{ OP_EQUALVERIFY} \\ \text{OP_CHECKSIGVERIFY} \end{array} \right.$
nLockTime = 0					

- where tw = now + two weeks (write it in two way, using block heights or Unix timestamps)
- If h is the current block height, put $tw = h + 2016$ as 2016 roughly translates to two weeks.
- After $T_{C_1 \rightarrow D}$ is accepted by the network, D immediately prepares and submits the following.

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{D \rightarrow F}$	$H(T_{C_1 \rightarrow D})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{D \rightarrow F}) \rangle \langle pk_{C_1} \rangle$	$x_1 \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 } \langle \text{HASH160}(pk_F) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right.$
nLockTime = tw_1					

- $T_{D \rightarrow F}$ is validated as follows. It is considered invalid if $tw_1 < tw$.
- OP_CHECKLOCKTIMEVERIFY in LScript for $T_{C_1 \rightarrow D}$ checks the above condition.
- If $tw_1 < tw$, the script execution continues. Otherwise, it halts and $T_{D \rightarrow F}$ deemed invalid.



Check Lock Time Verify (CLTV)

- After execution, if OP_CHECKLOCKTIMEVERIFY is satisfied, the $\langle tw \rangle$ that preceded it remains as the top item on the stack.
 - It is dropped, with OP_DROP, for correct execution of rest of the script.
-
- Therefore, with the above mechanism, C_1 can no longer **double-spend** the money as $T_{C_1 \rightarrow D}$ is immediately accepted by the network by spending the UTXO of $T_{B \rightarrow C_1, C_2}$.
 - Also, D cannot get the money before the two weeks - for $T_{D \rightarrow F}$ to be valid, D is forced to set nLocktime to two weeks !!

MultiSig

● m -out- n MultiSig :

- The multi sig scriptPubKey allows one to lock Bitcoins to the joint ownership of n public keys $\{Pk_1, \dots, Pk_n\}$ such that later the owners of any m out of these n public keys can come together to spend these Bitcoins.

txn	Input		Output	
	txid	scriptSig	UTXO	scriptPubkey
T_1			$x \text{ ₿}$	m $\langle PK_{A_1} \rangle \dots \langle PK_{A_n} \rangle$ n OP_CHECKMULTISIGVERIFY
T_2	$H(T_1)$	$\langle \text{Sign}_{sk_{A_{i_1}}}(T_2) \rangle \dots \langle \text{Sign}_{sk_{A_{i_m}}}(T_2) \rangle$		

- The T_2 transaction is validated by the network participants as follows: check if, $\text{Verify}_{pk_{A_{i_j}}}(\langle \text{Sign}_{sk_{A_{i_j}}}(T_2) \rangle, T_2) = \text{valid}$, for all j in $1 \leq j \leq m$.

A Fair Exchange Contract using Multi-Sig

- Suppose Bob accepts payment in bitcoin for selling a product
- Suppose Alice wants to buy a product from Bob
- But, they don't trust each other!!
 - Alice get's the product and keep her payment
 - Bob get's the payment but doesn't send the product
- Let's look at a first solution
- Alice locks the payment UTXO to the following lock script
`2 PK_Alice PK_Bob 2 OP_CHECKMULTISIG`
- **Pros**
- Bob won't get paid unless Alice gets the product
- Alice can't get the product and keep her payment.
- **Cons**
- Not useful if there is a dispute - such as product return
- Bob doesn't ship the product, and thus Alice money is lost (not to Bob)
- _____ An improved solution
- Alice and Bob invites an arbitrator - Max
- Alice locks the payment UTXO to the following lock script
`2 Alice_PK Bob_PK Max_PK 3 OP_CHECKMULTISIG`
- Max can arbitrate and decides who gets the UTXO if there is a dispute.

Pay to Script Hash

④ P2SH

- A P2SH LScript contains a hash of an another arbitrary-type lock script - called **redeem script**
- Consider a **Receiver** who is going to get ₿ from a **Sender**
- **Receiver** creates a redeem script (any type) - LS_{redeem}
- Hashes it: $h_{redeem} = HASH160(LS_{redeem})$
- Provides h_{redeem} to a **Sender**.

-
- **Sender** creates a P2SH lock script

$OP_HASH160 <h_{redeem}> OP_EQUAL$

-
- Later, when the **Receiver** wants to consume this UTXO, it provides the following two scripts
 - LS_{redeem}
 - ULS_{redeem}

● Validation

- Two scripts are combined and executed in two stages:
 - $LS_{redeem} \parallel OP_HASH160 <h_{redeem}> OP_EQUAL$
 - $ULS_{redeem} \parallel LS_{redeem}$

P2SH Addresses

● P2PKH Address

- $H1 = \text{SHA256}(\text{pk})$
- $H2 = \text{RIPEMD160}(H1)$
- $H3 = A \parallel H2$, where

$$A = \begin{cases} 0x00 & \text{main} \\ 0x6f & \text{test} \end{cases}$$

- $H4 = \text{SHA256}^{\text{double}}(H3)$
- $H5 = A \parallel H2 \parallel H4[1 : 4]$
- P2PKH address = Base58Encoding($H5$)

Address starts with 1 (Main)

Address starts with m/n (Test)

● P2SH Address

- $H1 = \text{SHA256}(\text{LS}_{\text{redeem}})$
- $H2 = \text{RIPEMD160}(H1)$
- $H3 = A \parallel H2$, where

$$A = \begin{cases} 0x05 & \text{main} \\ 0xC4 & \text{test} \end{cases}$$

- $H4 = \text{SHA256}^{\text{double}}(H3)$
- $H5 = H3 \parallel H4[1 : 4]$
- P2SH address = Base58Encoding($H5$)

Address starts with 3 (Main)

Address starts with 2 (Test)

Benefits of P2SH Address

- A **Receiver** might use a complex script to receive payment
- Instead of asking a **Sender** to use this complex script as lock script
 - Obtain a P2SH Address
 - give it the **Sender**
- Sender extracts Redeem Script hash $h_{redeem} \leftarrow$ P2SH address
- Creates the lock script as

`OP_HASH160 <h_redeem> OP_EQUAL`

- Complex scripts are replaced by shorter hash digest in the txn output
- Script can be encoded as address - sender's wallet don't need complex engineering to implement P2SH
- Shifts the burden of constructing the script to the recipient and not the sender
- Shifts the burden from UTXO mem pool to input (stored in block chain)
- Shifts the txn fee cost of a long txn from the sender to the receiver
- Receiver has to include the long redeem script to spend it

Pay to Script Hash: The flow

- Suppose, a user D requests user C_1 to transfer \mathbb{B} , locking it using a non-standard type lockscript !!

- Step-①: The user D prepares the following
 - Prepare a lockscript LS_{redeem}
 - Prepare the corresponding ULS_{redeem}
 - Prepare $P2SH(LS_{\text{redeem}})$
 - Finally send $P2SH(LS_{\text{redeem}})$ to C_1
- Step-②: C_1 extracts $h_{\text{redeem}} = \text{HASH160}(LS_{\text{redeem}}) \leftarrow P2SH(LS_{\text{redeem}})$

It then propose the following transaction

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1 \rightarrow D}$	$H(T_{B \rightarrow \{C_1, C_2\}})$	0	$\langle \text{Sign}_{sk_{C_1}}(m_{T_{C_1 \rightarrow D}, 0}) \rangle \langle pk_{C_1} \rangle$	$x_1 \mathbb{B}$	$OP_HASH160 \langle h_{\text{redeem}} \rangle OP_EQUALVERIFY$

- After $T_{C_1 \rightarrow D}$ is accepted by the network, the user D can spend the $x_1 \mathbb{B}$ later by pushing the following transaction to the network

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{D \rightarrow E}$	$H(T_{C_1 \rightarrow D})$	0	$ULS_{\text{redeem}} LS_{\text{redeem}}$	$x_2 \mathbb{B}$	$\{$ $OP_DUP OP_HASH160 \langle \text{HASH160}(pk_E) \rangle$ $OP_EQUALVERIFY OP_CHECKSIGVERIFY$ $\}$

OP_RETURN

- A data output lock script can be used to store data on the blockchain.

- Lock script of this type

OP_RETURN <data>

where data is limited to 40 bytes

-
- The data is a mostly hash digest (32-bytes) of arbitrary data
 - General structure of data= some prefix || hash digest
 - prefix identifies the application
 - For example: prefix = 0x444f4350524f46 for application <https://proofofexistence.com/>

-
- No unlock script correspond to OP_RETURN
 - UTXO is set to 0
 - Unlocking script containing OP_RETURN is considered invalid
 - One OP_RETURN as output per txn

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
T_{C_1}	$H(T_B \rightarrow \{C_1, C_2\})$	n	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1}) \rangle \langle pk_{C_1} \rangle$	$x_1 = 0$	OP_RETURN <data>

Script with IF/ELSE

- Suppose C_1 has paid D such that a timelock two weeks has been enabled before D can redeem the money.
 - What if after a week both C_1 and D come to an agreement that it is okay for D to get the money immediately.
 - But our method doesn't allow that - D is now forced to wait unnecessarily !!
 - _____ Here is a way out!
 - The Bitcoin Script supports composing conditions under "IF" and "ELSE".
 - C_1 can pay D using the following transaction:

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{C_1 \rightarrow D}$	$H(T_{B \rightarrow \{C_1, C_2\}})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1 \rightarrow D}) \rangle \langle pk_{C_1} \rangle$	$x_1 \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_IF} \\ 2 \langle pk_{C_1} \rangle \langle pk_D \rangle 2 \text{OP_CECKMULTISIGVERIFY} \\ \text{OP_ELSE} \\ \langle tw \rangle \text{OP_CHECKLOCKTIMEVERIFY} \text{OP_DROP} \\ \text{OP_DUP} \text{OP_HASH160} \langle \text{HASH160}(pk_D) \rangle \\ \text{OP_EQUALVERIFY} \text{OP_CHECKSIGVERIFY} \\ \text{OP_ENDIF} \end{array} \right.$
nLockTime = 0					

Hash-Locked

● Hash-Locked :

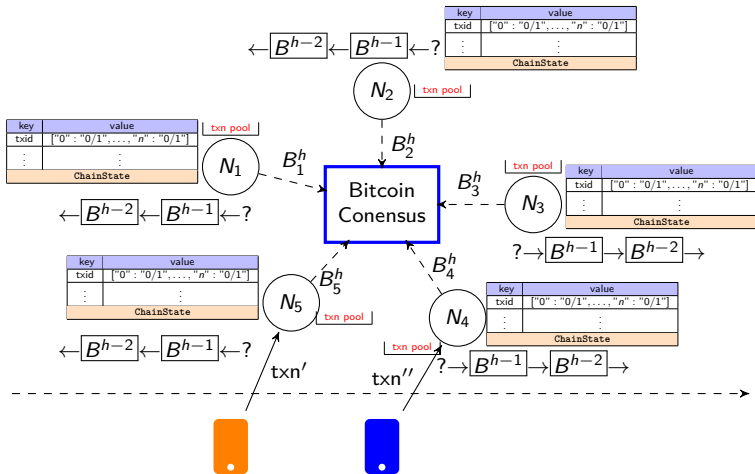
- This type of LScript allows us to lock Bitcoins by specifying a value such that any body, at a later point, can redeem the coins by providing a pre-image of the value under HASH160.
- In the following example, assume $\text{HASH160}(k) = h$.

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
T_{C_1}	$H(T_{B \rightarrow \{C_1, C_2\}})$	0	$\langle \text{Sign}_{sk_{C_1}}(T_{C_1}) \rangle \langle pk_{C_1} \rangle$	$x_1 \text{ ₿}$	$\text{OP_HASH160 } \langle h \rangle \text{ OP_EQUALVERIFY}$
nLockTime = 0					

- A user D will be able to redeem the above provided it has a pre-image of h

txn	Input			Unspent txn Output (UTXO)	
	txid	S.N.	scriptSig (UnLockScript)	Amount	scriptPubkey
$T_{D \rightarrow F}$	$H(T_{C_1})$	0	$\langle k \rangle$	$x_1 \text{ ₿}$	$\left\{ \begin{array}{l} \text{OP_DUP OP_HASH160 } \langle \text{HASH160}(pk_F) \rangle \\ \text{OP_EQUALVERIFY OP_CHECKSIGVERIFY} \end{array} \right.$
nLockTime = 0					

The Bitcoin Architecture



- 1 Bitcoin Network: Miners - N_1, \dots, N_5
- 2 Wallets:  
- 3 Distributed Ledger: Blockchain + ChainState



The Flow

● txn Creation

- Bitcoin transactions are created by wallets and pushed into the network
-

● txn Validation

- Transactions soon reach all full nodes in the network
 - Every node would validate received transactions
 - Validation requires each node to use [Bitcoin Core](#) software, and the special execution environment therein
 - The validated transactions are put into each node's [transaction pool](#)
 - Unconfirmed txn The txns that are present in the transaction pool of a node, but not in its blockchain.
-

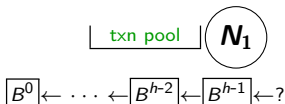
● txn Ordering

- The network now get together to reach a consensus on a set of unconfirmed txns next.
 - To this node, every node prepares a valid block containing a set of unconfirmed txns from its pool.
 - Network is said to reach a consensus if all nodes agree on the same block.
 - Confirmed txn A txn is tagged as confirmed if it is included into a valid block.
-

Bitcoin Miner's Local Database

Key	Value
$h(\text{txn}_{\ell_1})$	$["0" : \text{UTXO}_{\ell_1 0}, \dots, "r_1" : \text{UTXO}_{\ell_1 r_1}]$
$h(\text{txn}_{\ell_2})$	$["0" : \text{UTXO}_{\ell_2 0}, \dots, "r_2" : \text{UTXO}_{\ell_2 r_2}]$
\vdots	\vdots
$h(\text{txn}_{\ell_k})$	$["0" : \text{UTXO}_{\ell_k 0}, \dots, "r_k" : \text{UTXO}_{\ell_k r_k}]$

Chain State



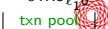
Block Chain

- ▶ N_1 represents a full node in Bitcoin
- ▶ N_1 's local database - "Bitcoin Ledger"
- ▶ "Ledger" \equiv "Blockchain" + "Chain State"
- ▶ Chain State is also called - World State of the Ledger
- ▶ Chain State is a key-value data base
 - ◇ key: $\text{txid} = h(\text{txn})$
 - ◇ value: txn outputs
 - ◇ outputs in red are considered spent
 - ◇ outputs in blue are considered un-spent

Transaction Validity

- ▶ Suppose, N_1 receives

$$\text{txnq}^* = \begin{cases} \text{Input0} : (h(\text{txn}_{\ell_1}), 0, \text{ULS}_{\text{UTXO}_{\ell_1 0}}) \\ \text{Output0} : \text{UTXO}_{q_0} \\ \text{Output1} : \text{UTXO}_{q_1} \end{cases}$$

- ▶ N_1 retrieves $\text{LS}_{\text{UTXO}_{\ell_1 0}} \leftarrow \text{Chain State}[h(\text{txn}_{\ell_1})]$
- ▶ Execute: $\text{ULS}_{\text{UTXO}_{\ell_1 0}} \parallel \text{LS}_{\text{UTXO}_{\ell_1 0}}$
- ▶ If no error, add txnq^* to 

Bitcoin Miner's Local Database

Transaction Validity

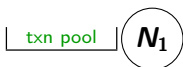
- Suppose, N_1 receives

$$\text{txnq}^* = \begin{cases} \text{Input0} : (h(\text{txn}_{\ell_1}), 0, \text{ULS}_{\text{UTXO}_{\ell_1 0}}) \\ \text{Output0} : \text{UTXO}_{q0} \\ \text{Output1} : \text{UTXO}_{q1} \end{cases}$$

- N_1 retrieves $\text{LS}_{\text{UTXO}_{\ell_1 0}} \leftarrow \text{Chain State}[h(\text{txn}_{\ell_1})(0)]$
- **Execute:** $\text{ULS}_{\text{UTXO}_{\ell_1 0}} \parallel \text{LS}_{\text{UTXO}_{\ell_1 0}}$
- If no error, add txnq^* to txn pool

- txnq^* is further relayed into the network by N_1
- Simultaneously, N_1 updates its local **Chain State** as follows:

Key	Value
$h(\text{txn}_{\ell_1})$	$["0" : \text{"UTXO}_{\ell_1 0}", \dots, "r_1" : \text{"UTXO}_{\ell_1 r_1}"]$
$h(\text{txn}_{\ell_2})$	$["0" : \text{"UTXO}_{\ell_2 0}", \dots, "r_2" : \text{"UTXO}_{\ell_2 r_2}"]$
\vdots	\vdots
$h(\text{txn}_{\ell_k})$	$["0" : \text{"UTXO}_{\ell_k 0}", \dots, "r_k" : \text{"UTXO}_{\ell_k r_k}"]$
$h(\text{txnq}^*)$	$["0" : \text{"UTXO}_{q0}", "1" : \text{"UTXO}_{q1}"]$



Chain State

Further Abstraction

- **Application** In our application, Bitcoin - a peer-to-peer currency, the primary components are:

- ① Bitcoins and some information associated to each of these coins, and
- ② A peer-to-peer network managing the Bitcoins

- We would like to view each instance of ① as an Asset, as follows !!

Key	Value
txid	Asset = [Asset Value, Owner , A Rule for Asset Update]

- **Asset Creation**

- An asset creation is done using a txn, and subsequently, the newly created asset, is placed in the database using the key = txn id

- **Asset Update**

- A transaction must be made to update an Asset.
- The "asset-update-rule" is used to validate the transaction before the update is applied.

- We have seen, how Bitcoin's blockchain-based peer-to-peer network securely manages the asset management - creation, and update !!

A Few Observations !!

- For most applications, we can think of them as consisting of the following components
 - a few types of assets that the application prescribes,
 - users in the application creates and updates these assets, by strictly following the application rules
 - A platform that helps to implement the above

-
- An example application: An anonymous feedback system with accountability
 - The application allows users to post feedbacks, anonymously
 - It also allows users to flag a already posted anonymous-feedback
 - If an anonymous feedback receives enough number of flags, the application must make the feedback non-anonymous by revealing the identity of the user who posted it
 - We denote the process of posting an anonymous feedback as making a transaction.

Key	Value
txid	Asset = [Feedback, Who Posted , Who Flagged = {}, Rulebook]

where Rulebook outlines when is precisely the identity of the user who posted the feedback can be revealed; other users can flag the feedback only once; or users are not allowed to flag their own feedbacks, etc.

- We take the process of an another user flagging this feedback as making a transaction that tries to update this asset!

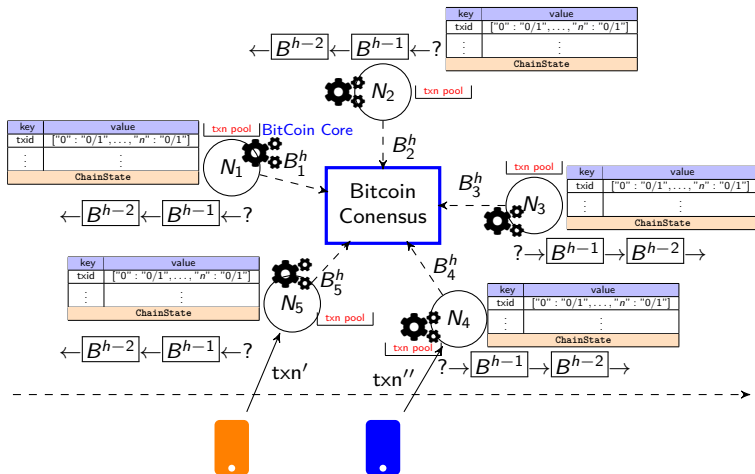
A Few Observations !!

- In Bitcoin's Blockchain-based peer-to-peer network, any one is allowed to join as a peer in the network
 - At any point of the time, no clarity on the total number of peers present in the network, or every peer's identity, etc.
 - This, in the presence of arbitrary number of malicious peers, makes the security of managing assets creation and updating them, an extremely difficulty process
-
- Can we make the problem a slightly easier by putting a restriction on peer enrolment ?
 - The answer is a yes!
 - With the knowledge of peer identities and the total number of peers present in the system, we may avoid, altogether, the expensive Bitcoin's distributed consensus and replace it by a well known distributed consensus algorithm !!
 - Though, we can continue to maintain a blockchain, chain of blocks, where each block contains valid transactions !!
-
- We call Bitcoin's blockchain-based peer-to-peer network as **Permission-less**
 - By enforcing a restriction on the peer enrolment, we call the resulting blockchain as **Permissioned Blockchain**.

Bitcoin's Architecture as a Blockchain-based Distributed Application

- ① Application It can be described as users **creating** and **updating** a few types of **assets** and they do so, by proposing different types of **transactions**.
 - ② Distributed Database The database for all assets (as **key-value** pairs), and all the transactions, are implemented in distributed fashion among multiple servers (peers).
 - ③ Distributed Consensus The peers take part in a distributed algorithm, at a regular interval, to reach a consensus on a set of newly proposed transactions for their,
 - validation, (as per the prescribed **rules** of the application)
 - ordering, and
 - execution (leading to creation and update of assets)
-

The Bitcoin Architecture



- 1 Bitcoin Network: Miners - N_1, \dots, N_5
- 2 Wallets:  
- 3 Distributed Ledger: Blockchain + ChainState

Bitcoin's Architecture as a Blockchain-based Distributed Application

- ① **Application** It can be described as users **creating** and **updating** a few types of **assets** and they do so, by proposing different types of **transactions**.
- ② **Distributed Database** The database for all assets (as **key-value** pairs), and all the transactions, are implemented in distributed fashion among multiple servers (peers).
- ③ **Distributed Consensus** The peers take part in a distributed algorithm, at a regular interval, to reach a consensus on a set of newly proposed transactions for their,
 - validation, (as per the prescribed **rules** of the application)
 - ordering, and
 - execution (leading to creation and update of assets)

- Bitcoin's Blockchain-based distributed framework is Permission-less as it allows any body to be a peer !! The expensive distributed consensus mechanism of Bitcoin is famously called as "**Proof-of-Work**"

-
- If we are to limit the entry of peers, then the underlying Blockchain-based distributed framework would be termed as Permissioned.
 - Clearly, a permissioned blockchain-based distributed framework doesn't required as complex a consensus mechanism as the Proof-of-Work !!

-
- Enterprise Blockchain Framework: Limiting the entry of peers by allowing them from a single organisation
 - Consortium Blockchain Framework: Limiting the entry of peers by only allowing them from a pre-approved set of organisations (not trusted by each other)

A Modular System to Deploy Permissioned-Blockchain Frameworks

- Recall the application: Anonymous feedback system with accountability
 - The application allows users to post feedbacks, anonymously
 - It also allows users to flag a already posted anonymous-feedback
 - If an anonymous feedback receives enough number of flags, the application must make the feedback non-anonymous by revealing the identity of the user who posted it

- Asset Type

Key	Value
txid	Asset = [Feedback, Who Posted , Who Flagged = {}]

where txid is the transaction

identifier of the txn that created this asset in the first place.

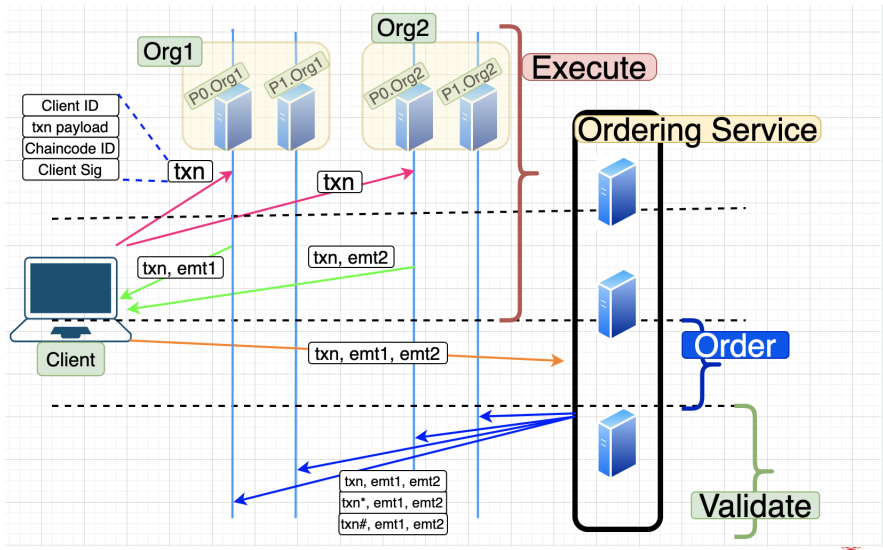
- Rulebook A Rulebook outlines how a transaction must be validated before it is allowed to create an asset or modify an existing one.
- Network Configuration We might insist that, the distributed network must have peers allowed only from two pre-approved organisations.

-
- Can we think of a system which allows us to implement, the above application, the following way !!
 - Step-① Put in place, a peer-to-peer to network, where each peers is from one of the two pre-approved organisation
 - Step-② Import the **Application Rulebook** into each of these peers, where the rule book prescribed the asset types, and the rules that must be followed to update existing assets !!
 - Step-③ Configure all peers with the same distributed consensus algorithm that they can use to validate, order and execute newly proposed transactions!

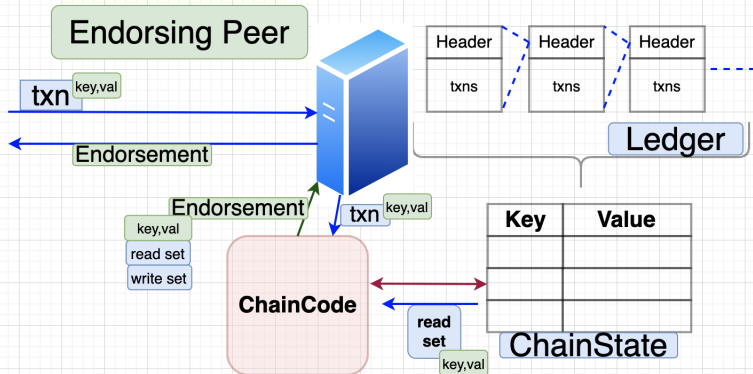
Hyperledger Fabric

- Hyperledger Fabric <https://www.hyperledger.org/use/fabric> is an open-source system for deploying and operating Permissioned Blockchain-based distributed frameworks
 - The Permissioned Blockchain-based distributed frameworks, in turn, run dedicated distributed applications.
 - Fabric is one of the Hyperledger <https://www.hyperledger.org/> projects hosted by the Linux Foundation
-

Hyperledger Fabric: Architecture and Flow of the Deployed Blockchain-based Distributed System



Peer Architecture: Endorsing Peer



Peer Architecture: Committing Peer

