What is Blockchain ?

Blockchain is a distributed Ledger which is

- Immutable
- Tamperproof
- Decentralized
- Trust-less

The Byzantine Generals Problem

m = 0 → no traitors, each lieutenant obeys

m > 0 → each lieutenant's final choice comes from the majority of all lieutenant's choices,

For m< n/3

Algorithm OM(0).

(1) The commander sends his value to every lieutenant.
(2) Each lieutenant uses the value he receives from the commander, or uses the value RETREAT if he receives no value.

Algorithm OM(m), $m > 0$.

(1) The commander sends his value to every lieutenant.
(2) For each i, let v_i be the value Lieutenant i receives from the commander, or else be RETREAT if he receives no value. Lieutenant i acts as the commander in Algorithm OM($m - 1$) to send the value v_i to each of the $n - 2$ other lieutenants.
(3) For each i, and each $j \neq i$, let v_j be the value Lieutenant i received from Lieutenant j in step (2) (using Algorithm OM($m - 1$)), or else RETREAT if he received no such value. Lieutenant i uses the value *majority*$(v_1, \ldots, v_{n-1})$.

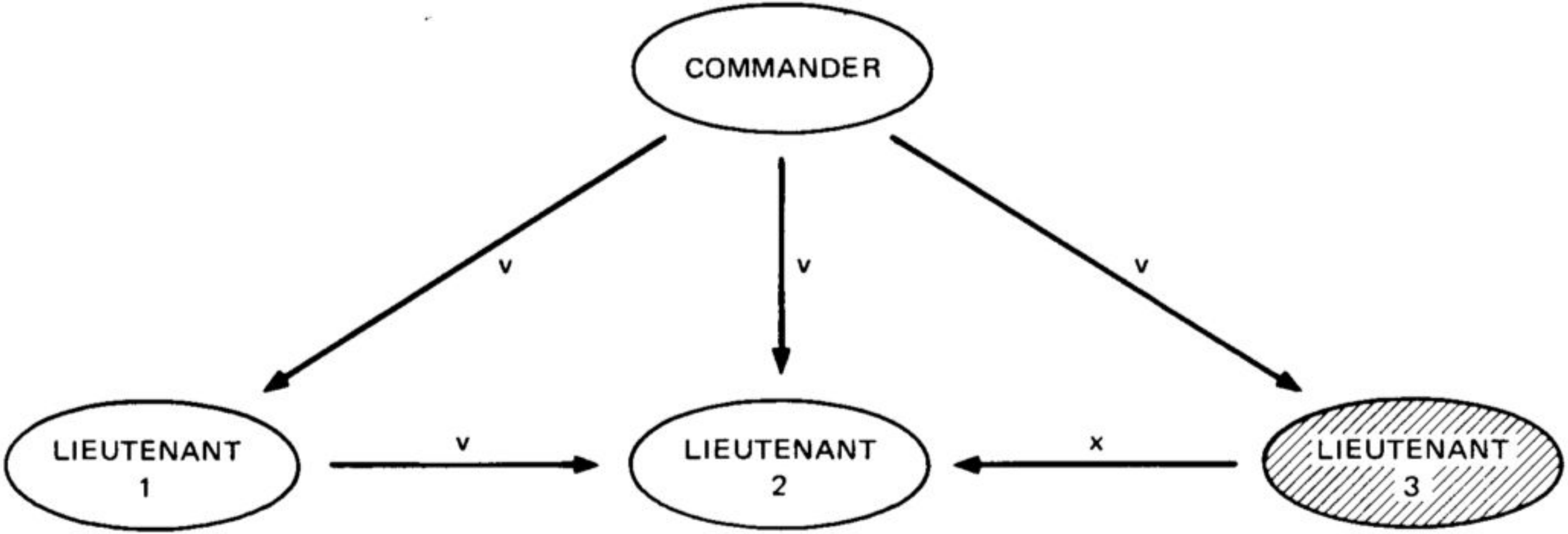
1. Commander sends v to all Lieutenants

2. L1 sends v to L2 | L3 sends x to L2

3. L2 $\leftarrow$ $majority(v,v,x) == v$

OM(1): Lieutenant 3 is a traitor

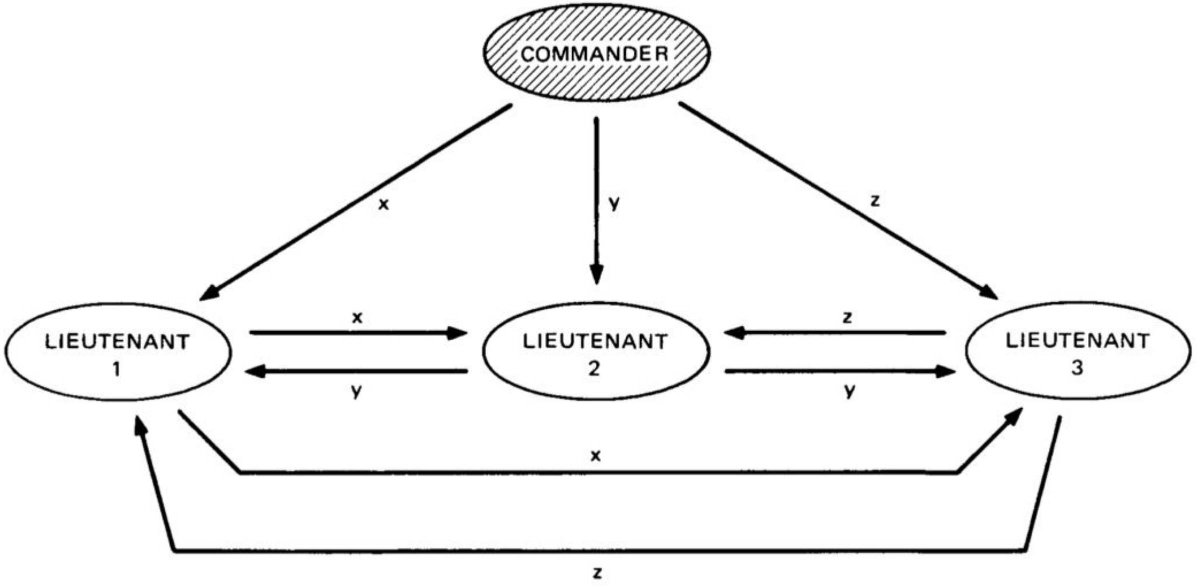
—L2 point of view



OM(1): Commander is a traitor

1. Commander sends *x*, *y*, *z* to L1, L2, L3 respectively
2. L1 sends *x* to L2, L3 | L2 sends *y* to L1, L3 | L3 sends *z* to L1, L2
3. L1 ← majority(x,y,z) | L2 ← majority(x,y,z) | L3 ← majority(x,y,z)

Goal is for the majority of th lieutenants to choose the ***same*** *decision, not a specifi one!*



Solution is not efficient enough

1] it has constraints, that less than one-third of the network is dishonest,

2] It is slow for a large network. $O(n^2)$

To prove the correctness of the algorithm OM(m) for arbitrary m, we first prove the following lemma.

LEMMA 1. *For any m and k, Algorithm OM(m) satisfies IC2 if there are more than $2k + m$ generals and at most k traitors.*

PROOF. The proof is by induction on m. IC2 only specifies what must happen if the commander is loyal. Using A1, it is easy to see that the trivial algorithm OM(0) works if the commander is loyal, so the lemma is true for $m = 0$. We now assume it is true for $m - 1$, $m > 0$, and prove it for m.

In step (1), the loyal commander sends a value v to all $n - 1$ lieutenants. In step (2), each loyal lieutenant applies OM($m - 1$) with $n - 1$ generals. Since by hypothesis $n > 2k + m$, we have $n - 1 > 2k + (m - 1)$, so we can apply the induction hypothesis to conclude that every loyal lieutenant gets $v_j = v$ for each loyal Lieutenant j. Since there are at most k traitors, and $n - 1 > 2k + (m - 1) \geq 2k$, a majority of the $n - 1$ lieutenants are loyal. Hence, each loyal lieutenant has $v_i = v$ for a majority of the $n - 1$ values i, so he obtains $majority(v_1, \ldots, v_{n-1}) = v$ in step (3), proving IC2. □

The following theorem asserts that Algorithm OM(m) solves the Byzantine Generals Problem.

THEOREM 1. *For any m, Algorithm OM(m) satisfies conditions IC1 and IC2 if there are more than $3m$ generals and at most m traitors.*

PROOF. The proof is by induction on m. If there are no traitors, then it is easy to see that OM(0) satisfies IC1 and IC2. We therefore assume that the theorem is true for OM$(m-1)$ and prove it for OM(m), $m > 0$.

We first consider the case in which the commander is loyal. By taking k equal to m in Lemma 1, we see that OM(m) satisfies IC2. IC1 follows from IC2 if the commander is loyal, so we need only verify IC1 in the case that the commander is a traitor.

There are at most m traitors, and the commander is one of them, so at most $m-1$ of the lieutenants are traitors. Since there are more than $3m$ generals, there are more than $3m-1$ lieutenants, and $3m-1 > 3(m-1)$. We may therefore apply the induction hypothesis to conclude that OM$(m-1)$ satisfies conditions IC1 and IC2. Hence, for each j, any two loyal lieutenants get the same value for v_j in step (3). (This follows from IC2 if one of the two lieutenants is Lieutenant j, and from IC1 otherwise.) Hence, any two loyal lieutenants get the same vector of values $v_1, \ldots, v_{n-1}$, and therefore obtain the same value *majority*$(v_1, \ldots, v_{n-1})$ in step (3), proving IC1. $\square$

Solution through Blockchain

Hashing Functions

Taking an input string of any length and giving out an output of a fixed length.

Properties

1. Deterministic : Every time you parse through a particular input through a hash function you will always get the same result.

2. Quick: O(1)

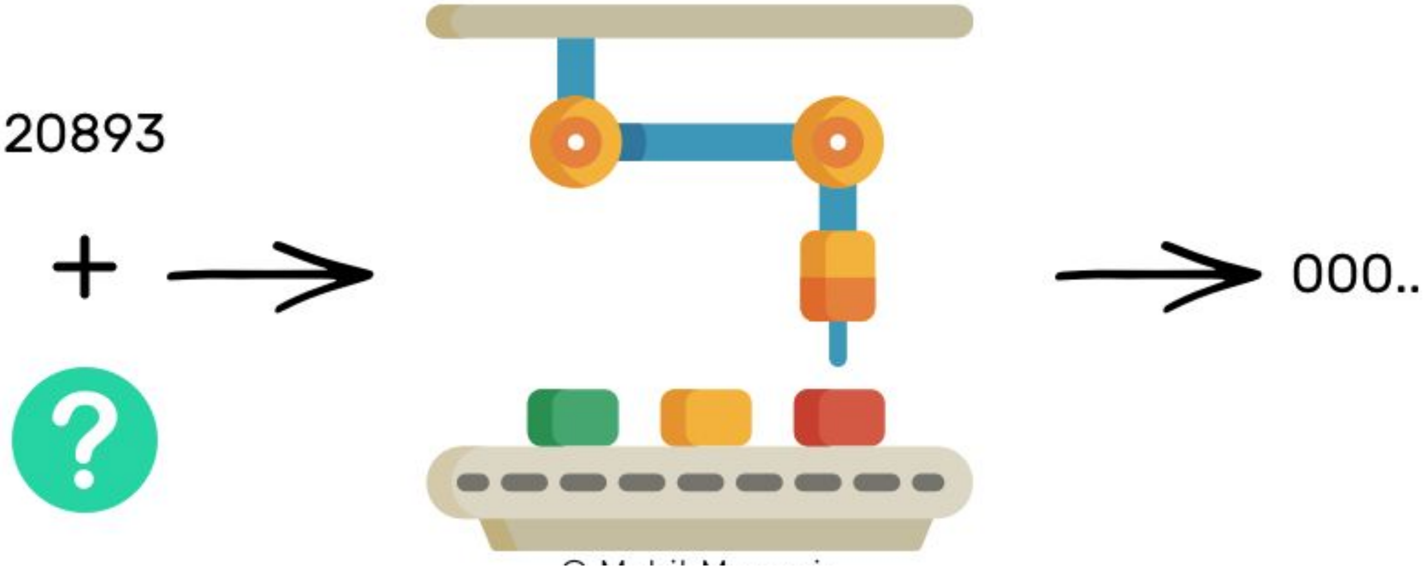
3. Given H(A) it is infeasible to determine A, where A is the input and H(A) is the output hash

4. They are unpredictable

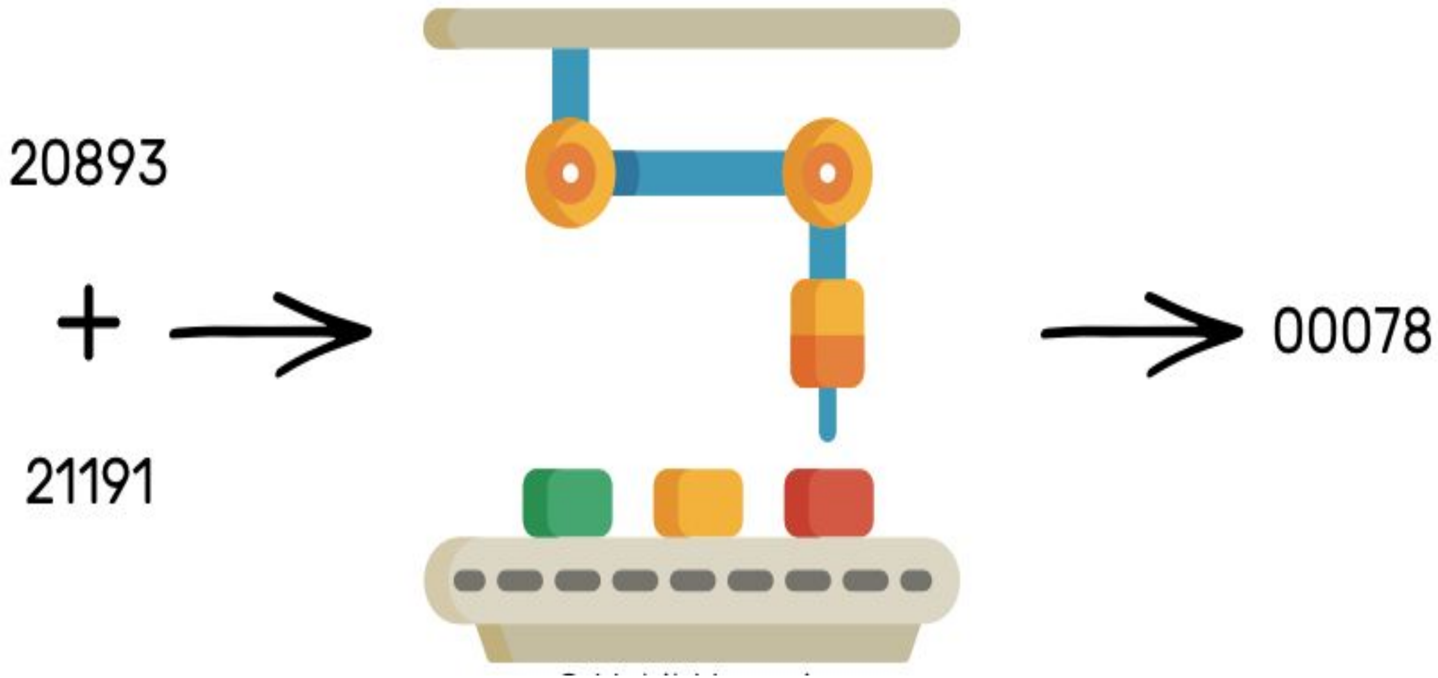
```python
    def hash(self, block):
        """
        Create a SHA-256 hash of a block
        """
        # We must make sure that the Dictionary is Ordered, or we'll have inconsistent hashes
        block_string = json.dumps(block, sort_keys=True).encode()

        return hashlib.sha256(block_string).hexdigest()

```

How blockchain uses hash

Can you figure out a number that when added to the number in the first box and fed to the machine will give us a word that starts with three leading zeroes?

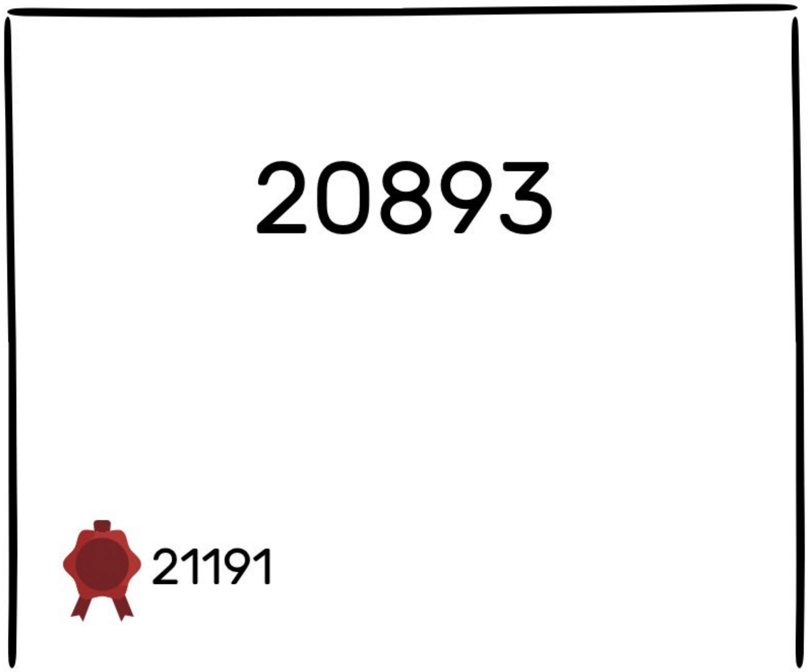


After several thousand attempts ...



- Finding out this number is called **mining**
- Hashing helps in **Zero-knowledge Proof**

Proof of Work



We got the sealing number! Which is the proof that work has happened to find that number.

Moreover it is easy to verify this anytime in future!

This number is called **NONCE**

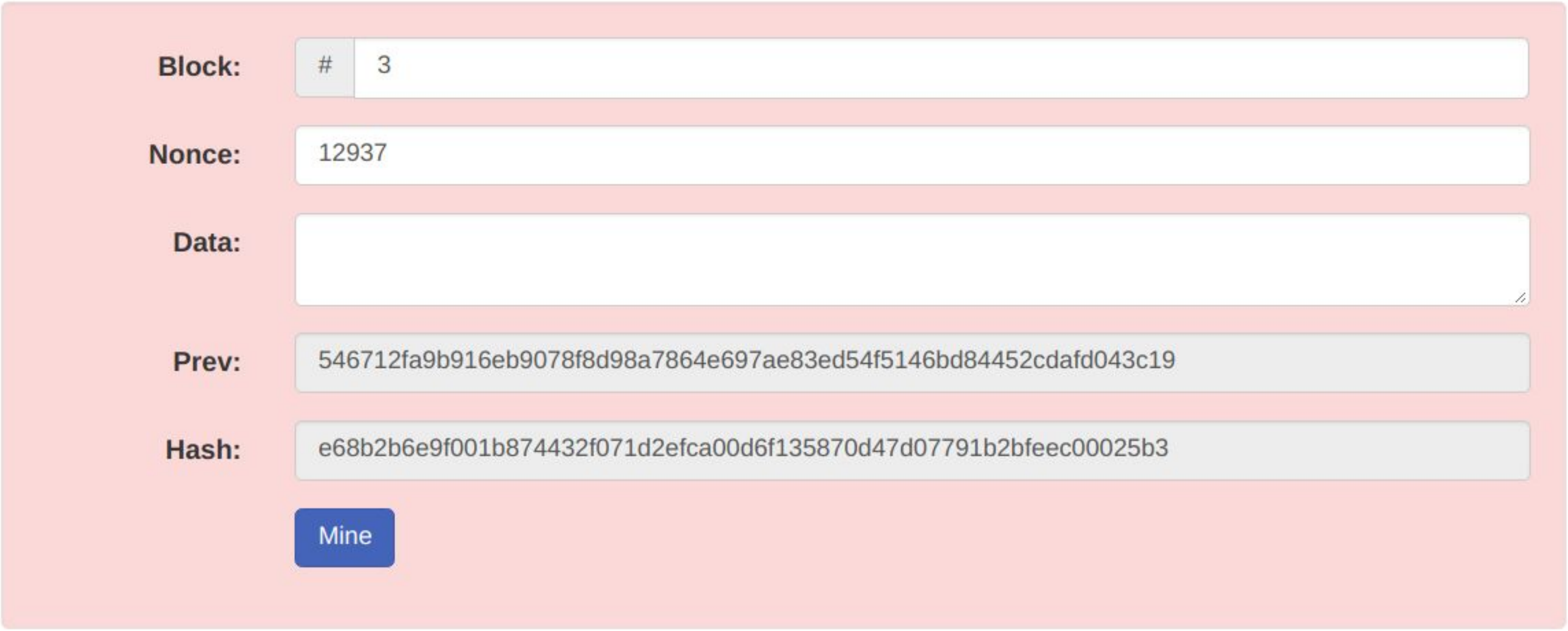
```python
def proof_of_work(self):
    """
    Proof of work algorithm
    """
    last_block = self.chain[-1]
    last_hash = self.hash(last_block)

    nonce = 0
    while self.valid_proof(self.transactions, last_hash, nonce) is False:
        nonce += 1

    return nonce

```

Mining

- Hard!
- Other miner Verify (easy) once a miner calculate and announce a NONCE
- Mining Bitcoin now consumes more than 30 terawatt-hours of power globally, which is higher than the individual energy usage of 159 countries

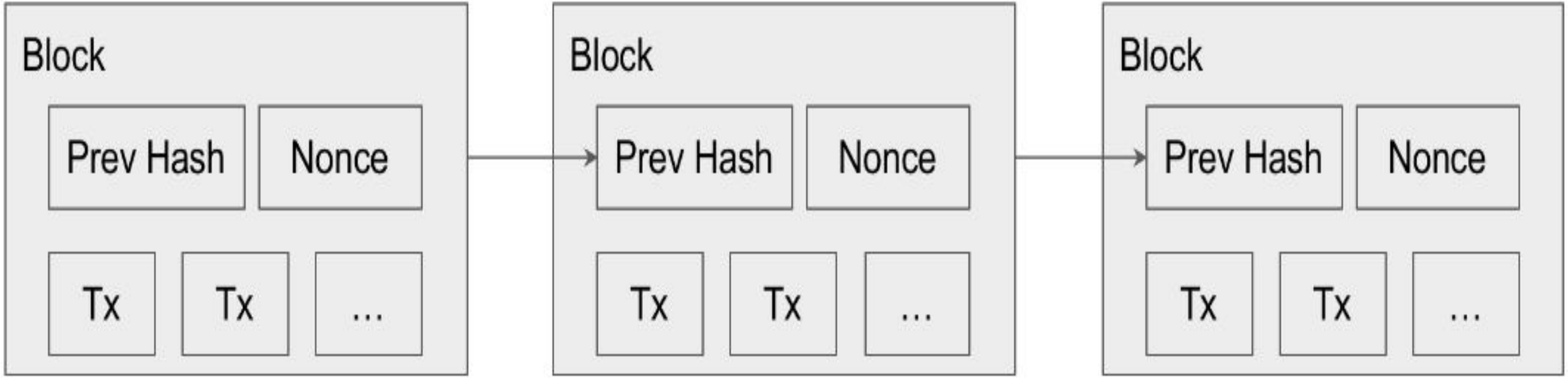
```python
    def valid_proof(self, transactions, last_hash, nonce, difficulty=MINING_DIFFICULTY):
        """
        Check if a hash value satisfies the mining conditions. This function is used within the proof_of_work function.
        """
        guess = (str(transactions)+str(last_hash)+str(nonce)).encode()
        guess_hash = hashlib.sha256(guess).hexdigest()
        return guess_hash[:difficulty] == '0'*difficulty
```

What does a block contain?



```python

    def create_block(self, nonce, previous_hash):
        """
        Add a block of transactions to the blockchain
        """
        block = {'block_number': len(self.chain) + 1,
                 'timestamp': time(),
                 'transactions': self.transactions,
                 'nonce': nonce,
                 'previous_hash': previous_hash}

        # Reset the current list of transactions
        self.transactions = []

        self.chain.append(block)
        return block
```

Linking the blocks in a chain!

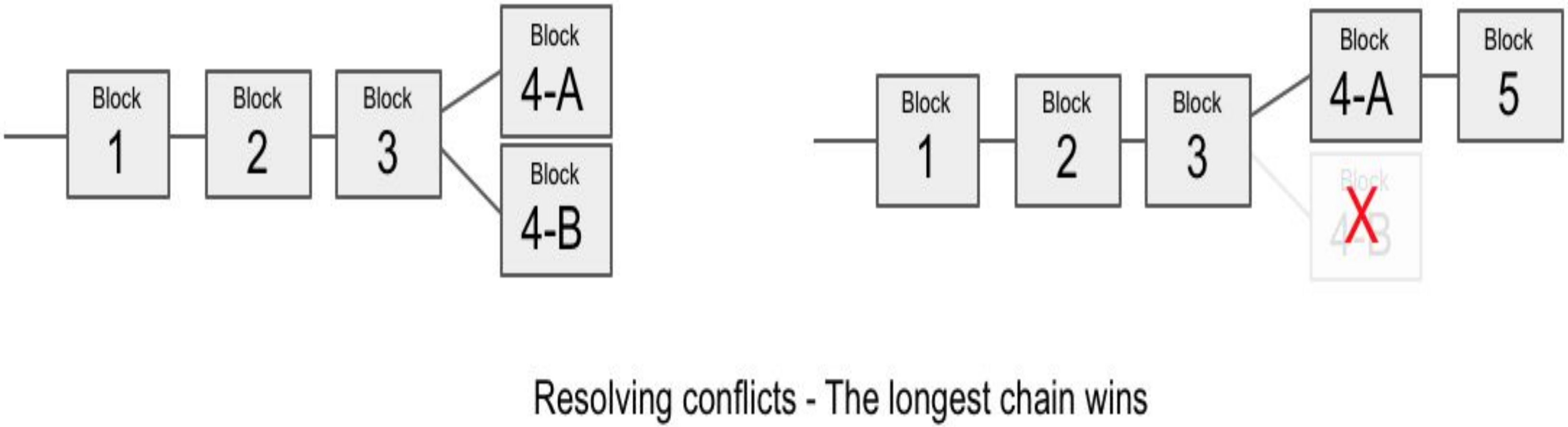


Blocks are chained together using the previous block's hash to form a Blockchain.

Changing one block Changes the chain completely

```python
    def valid_chain(self, chain):
        """
        check if a bockchain is valid
        """
        last_block = chain[0]
        current_index = 1

        while current_index < len(chain):
            block = chain[current_index]
            #print(last_block)
            #print(block)
            #print("\n-----------\n")
            # Check that the hash of the block is correct
            if block['previous_hash'] != self.hash(last_block):
                return False

            # Check that the Proof of Work is correct
            #Delete the reward transaction
            transactions = block['transactions'][:-1]
            # Need to make sure that the dictionary is ordered. Otherwise we'll get a different hash
            transaction_elements = ['sender_address', 'recipient_address', 'value']
            transactions = [OrderedDict((k, transaction[k]) for k in transaction_elements) for transaction in transactions]

            if not self.valid_proof(transactions, block['previous_hash'], block['nonce'], MINING_DIFFICULTY):
                return False

            last_block = block
            current_index += 1

        return True
```

Consensus



Note that: This is capable of tolerating 50% of “traitors”, which is higher as compared to 33% given in Byzantine Generals solution

```python
    def resolve_conflicts(self):
        """
        Resolve conflicts between blockchain's nodes
        by replacing our chain with the longest one in the network.
        """
        neighbours = self.nodes
        new_chain = None

        # We're only looking for chains longer than ours
        max_length = len(self.chain)

        # Grab and verify the chains from all the nodes in our network
        for node in neighbours:
            print('http://' + node + '/chain')
            response = requests.get('http://' + node + '/chain')

            if response.status_code == 200:
                length = response.json()['length']
                chain = response.json()['chain']

                # Check if the length is longer and the chain is valid
                if length > max_length and self.valid_chain(chain):
                    max_length = length
                    new_chain = chain

        # Replace our chain if we discovered a new, valid chain longer than ours
        if new_chain:
            self.chain = new_chain
            return True

        return False
```

References

- Code from : Dumb coin
- Hash Calculator : https://www.tools4noobs.com/online_tools/hash/
- Algo of BGP : The Byzantine Generals Problem - People @ EECS at UC Berkeley
- Byzantine Generals Problem video : https://www.youtube.com/watch?v=_MwqAaVweJ8
- Photos from blog1 and blog2

Thank You