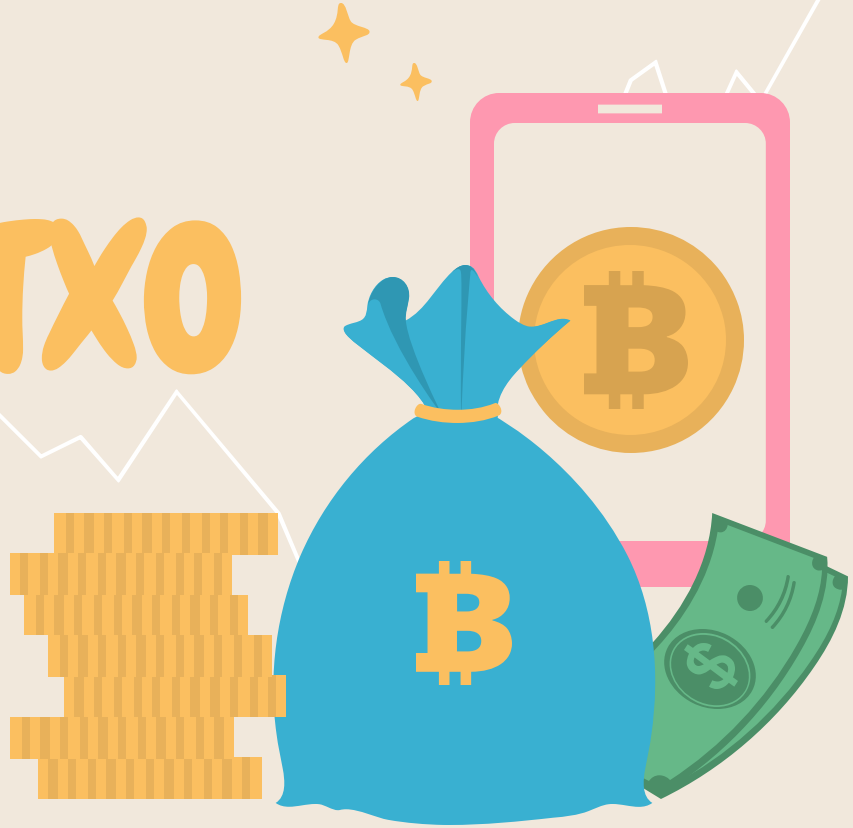
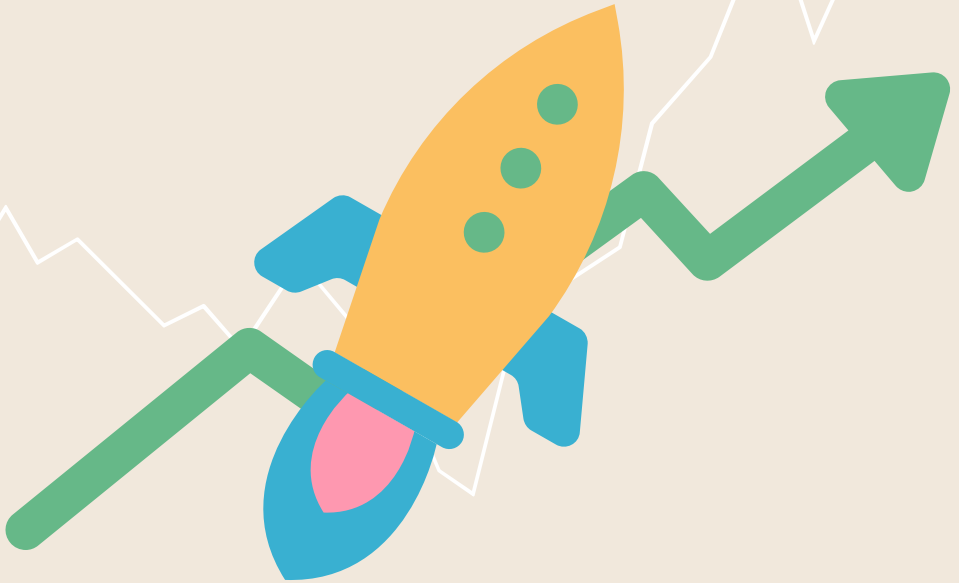
Comp4142 UTXO



20042635d Lau Yiu Hei
20048665d Yu Ngo Ting
20051248d Leung Kit Chuen
20051835d Mak Ngai Long
20056304d Lai Wing Ho

01
Blockchain Prototype



Blockchain Prototype

Design:

- One class for ‘Block’, one class for ‘Blockchain’
- ‘Blockchain’ class store and link all ‘Block’ by a List
- Methods are mainly implemented in ‘Blockchain’
    - Create a new ‘Block’
    - Hash function
    - Difficulty calculation
    - PoW algorithm
    - Nodes manipulation

Blockchain Prototype

'Block' class

```python
class Block(TypedDict):
    """Block class"""
    index: int
    timestamp: str
    transactions: List[Dict[str, Any]]
    proof: int
    previous_hash: str
    current_hash: str
    difficulty: int
    nonce: int
    markle_root: str
    data: str
```

- transactions: a list of dictionary
- proof: result of PoW algorithm

'Blockchain' class

```python
class Blockchain:
    """The main blockchain class."""
    TARGET_TIME = 10

    def __init__(self) -> None:
        self.transactions: List[Dict[str, Any]] = []
        self.chain: List[Block] = []    # The chain container

        self.nodes: Set[str] = set()
        self.node_id: str = str(uuid4()).replace('-', '')

        # Create the genesis block
        print("Creating genesis block")
        self.create_block(proof=1, prev_hash="genesis")
```

```python
def create_block(self, proof: int=None, prev_hash: str=None) -> Block:
    """Create a new Block in the Blockchain
    the Block should be immutable.

    :param proof: <int> The proof given by the Proof of Work a
    :param previous_hash: <str> Hash of the previous Block
    :return: <Block> A new Block
    """
    block = Block(
            index = len(self.chain),
            timestamp = str(datetime.now()),
            transactions = self.transactions,
            proof = proof or self.proof_of_work(self.last_
            previous_hash = prev_hash or self.hash(self.last_block),
            current_hash = self.calculate_current_hash(proof,prev_hash),
            difficulty = self.calculate_difficulty(),
            nonce = self.calculate_nonce(proof, prev_hash),
            merkle_root = self.calculate_merkle_root(),
            data = self.calculate_data()
        )
```

```python
@staticmethod
def hash(block: Block) -> str:
    """Creates a SHA-256 hash of a Block.

    :param block: <Block>
    :return: <str> Hash
    """
    # The dictionary has to be Ordered, or we'll have inconsistent hashes.
    encoded_block = json.dumps(block, sort_keys=True).encode()
    return sha256(encoded_block).hexdigest()
```

02

Mining and UTXO

Mining and UTXO

Proof-of-Work algorithm

- Input parameter: the last block
- Initialize 'proof' to 0
- Validate proof by the hashed last block, proof, and difficulty
  - Hashing them -> check the number of leading zero
  - If equal to difficulty -> pass validation
- Return proof

```python
def proof_of_work(self, last_block: Block) -> int:
    """Proof of Work Algorithm:
    - Find a number p' such that hash(pp') contains leading 4 zeroes, where p is the previous p'
    - p is the previous proof, and p' is the new proof
    :param last_block: <Block> The Last Block
    :return: <int> Proof
    """
    last_proof = last_block["proof"]
    last_hash = self.hash(last_block)
    difficulty = self.calculate_difficulty()

    proof = 0
    while self.validate_proof(last_proof, proof, last_hash, difficulty) is False:
        proof += 1

    return proof
```

```python
@staticmethod
def validate_proof(last_proof: int, proof: int, last_hash: str, difficulty: int) -> bool:
    """Validates the Proof

    :param last_proof: <int> Previous Proof
    :param proof: <int> Current Proof
    :param last_hash: <str> The hash of the Previous Block
    :return: <bool> True if correct, False if not.
    """
    guess: bytes = f'{last_proof}{proof}{last_hash}'.encode()
    guess_hash = sha256(guess).hexdigest()
    return guess_hash[:difficulty] == "0" * difficulty
```

Mining and UTXO

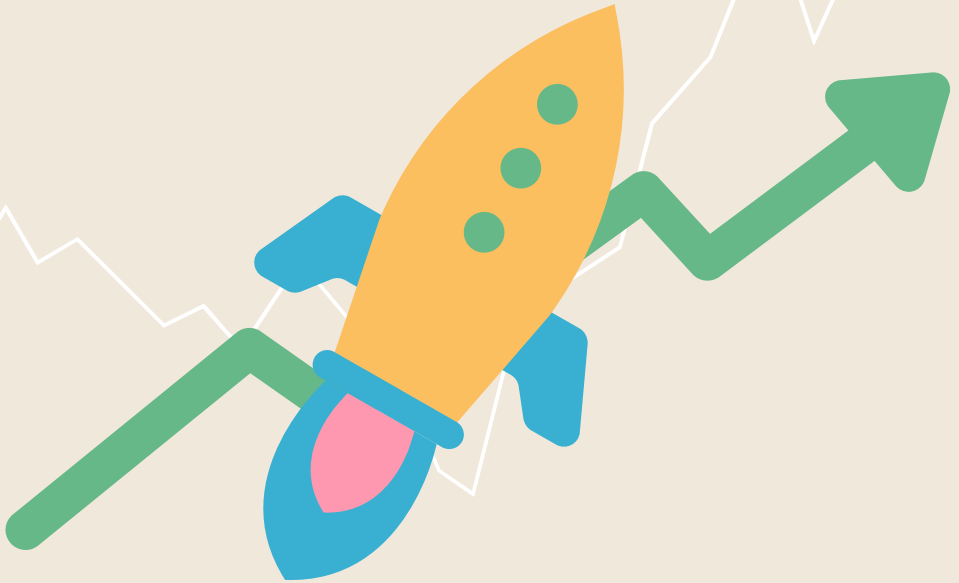
Dynamic difficulty

- Set target time to 10
- Get time taken to create the last block
  - Timestamp of last block - timestamp of second last block
- If time taken > target time, difficulty –1, otherwise +1

```python
def calculate_difficulty(self) -> int:
    """Calculate the difficulty of the next block based on the time taken to generate the previous blocks.
    :return: <int> Difficulty
    """
    if len(self.chain) < 2:
        return 4  # Initial difficulty

    # Calculate the time taken to generate the last block
    last_block = self.chain[-1]
    second_last_block = self.chain[-2]
    time_taken = datetime.fromisoformat(last_block['timestamp']) - datetime.fromisoformat(second_last_block['timestamp'])

    # If the time taken is greater than the target time, decrease the difficulty
    if time_taken.total_seconds() > self.TARGET_TIME:
        return last_block['difficulty'] - 1
    # If the time taken is less than the target time, increase the difficulty
    elif time_taken.total_seconds() < self.TARGET_TIME:
        return last_block['difficulty'] + 1

    return last_block['difficulty']
```

03

Transaction



Goal3: Transaction-P2PKH Transactions

Transaction Creation ('new_transaction' Method)

- This method process transaction creation.
- 1)sender, 2)recipient addresses, 3)amount, and 4)sender's private key ('SK') and 5)public key('PK')
- Generates a transaction dictionary with cryptographic hash values.
- Secures sensitive information and validates transaction authenticity

```python
def new_transaction(self, transaction: dict, SK, PK) -> int:

    private_key = pkcs1_15.new(RSA.import_key(SK))
    tx = transaction.copy()
    tx['hashed'] = SHA256.new((str(tx["sender"]) + str(tx["recipient"]) + str(tx["amount"])).encode()
    tx['signature'] = private_key.sign(tx['hashed'])

    # Test if both PK and SK can verify the transaction
    if (self.verify_transaction(tx, SK) and self.verify_transaction(tx, PK)):
        self.transactions.append(tx)
        return self.last_block['index'] + 1
    else:
        return self.last_block['index']
```

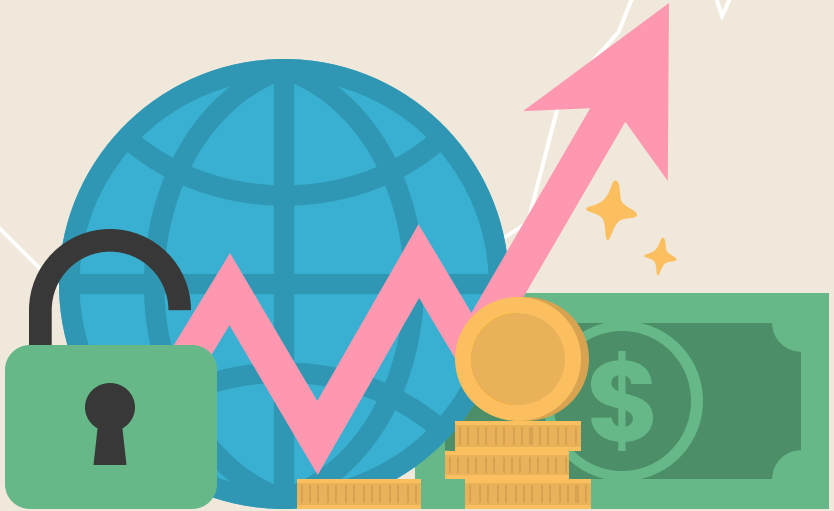
Verification Method ('verify_transaction)

- **Inputs:** Transaction dictionary, sender's 'SK' and 'PK'.
- **Process:** Creates a signature by signing the hashed transaction value with 'SK'.
- **Validation:** Uses the signature to verify against the hashed value using the sender's public key.

```python
# Use sender's public/secret key to verify the transaction
def verify_transaction(self, tx: Dict[str, Any], key) -> bool:

    key = pkcs1_15.new(RSA.import_key(key))
    try:
        key.verify(tx["hashed"], tx["signature"])
        return True
    except:
        print("Invalid Key!")
        return False
```

04

Network



```python
@app.route('/blocks', methods=['POST'])
def receive_block():
    block = request.get_json()
    if blockchain.validate_block(block):
        blockchain.chain.append(block)
        return "Block added to the chain", 200
    else:
        return "Invalid block", 400
```

```python
@app.route('/blocks', methods=['GET'])
def send_blocks():
    return json.dumps(blockchain.chain)
```

```python
@app.route('/balance', methods=['GET'])
def get_wallet_balance():
    balance = wallet.get_balance()
    return jsonify({'balance': balance}), 200
```

```python
@app.route('/wallet/address', methods=['GET'])
def get_wallet_address():
    address = wallet.public_key_to_address(wallet.public_key)
    return jsonify({'address': address}), 200
```

```python
@app.route('/create_transaction', methods=['POST'])
def create_and_add_transaction():
    data = request.get_json()
    recipient = data.get('recipient')
    amount = data.get('amount')
    print(recipient,amount)
    if recipient is None or amount is None:
        return jsonify({'error': 'Missing recipient or amount'}), 400
    transaction = wallet.create_transaction(recipient, amount)
    blockchain.new_transaction(transaction, wallet.private_key, wallet.public_key)
    return jsonify({'transaction': transaction}), 200
```

Goal4: Network

/blocks(POST):

- This endpoint is used to receive a new block from other nodes in the network and check the block is valid or not.

/blocks(GET):

- This endpoint is used to retrieve the entire chain of blocks from the blockchain network.

/balance(GET):

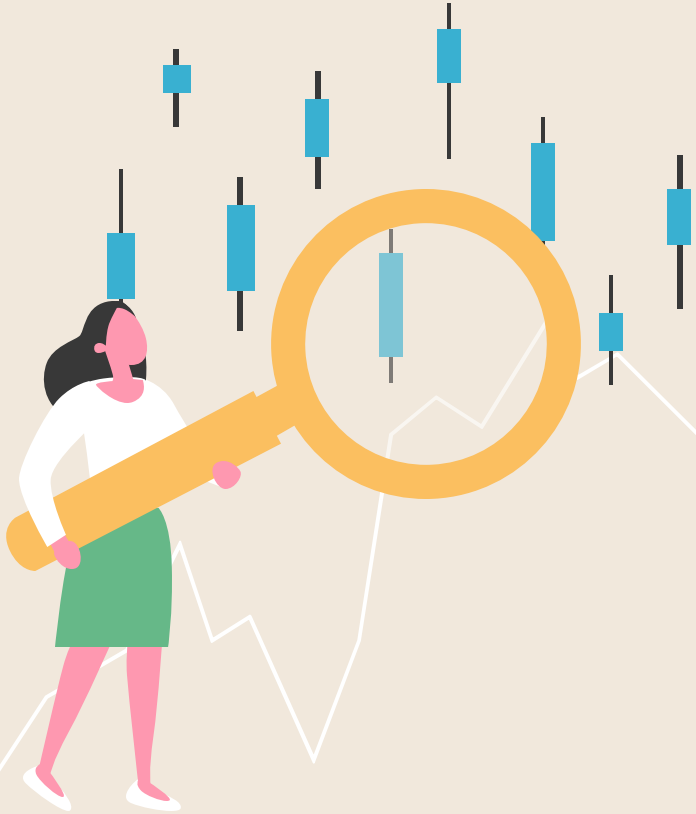
- This endpoint retrieves the balance of the wallet associated with the current user.

/wallet/address(GET):

- This endpoint retrieves the public address of the wallet associated with the current user.

/create_transaction(POST):

- This endpoint is used to create a new transaction on the blockchain.



05

Storage

Goal5: Storage

Latest State of the Blockchain in Memory:

- **'self.chain':** Holds the entire blockchain, preserving a complete record of blocks.
- **'self.nodes':** Tracks participating nodes within the blockchain network.
- **'self.node_id':** Provides a unique identifier for each node, aiding in node recognition.

```python
class Blockchain:
    """The main blockchain class."""
    TARGET_TIME = 10

    def __init__(self) -> None:
        self.transactions: List[Dict[str, Any]] = []
        self.chain: List[Block] = []    # The chain container

        self.nodes: Set[str] = set()
        self.node_id: str = str(uuid4()).replace('-', '')

        # Create the genesis block
        print("Creating genesis block")
        self.create_block(proof=1, prev_hash="genesis")
```

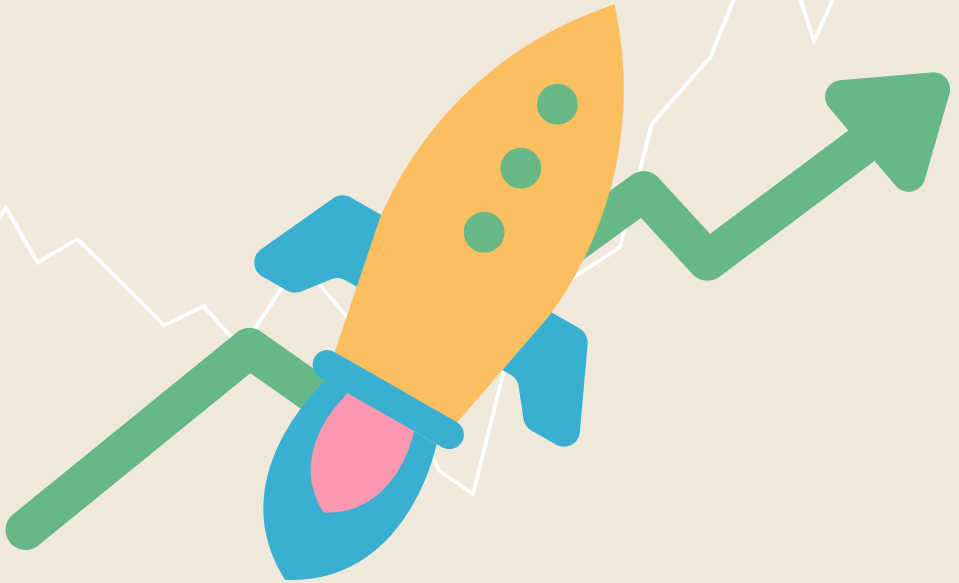
Storing Transactions (UTXO) in a Transaction Pool:

- **'self.transactions':** Functions as a transaction pool, temporarily housing pending transactions.
- **'new_transaction' method**: Manages Unspent Transaction Outputs (UTXOs) until they are processed into blocks.

```python
def new_transaction(self, sender_address: str, recipient_address: str, amount: float, SK) -> int:
    transaction = {
        'sender': sender_address,
        'recipient_address': recipient_address,
        'amount': amount,
        'hashed_value': SHA256.new((sender_address + recipient_address + str(amount)).encode()),
        'signature': ''
    }
    if self.verify_transaction(transaction, SK):
        self.transactions.append(transaction)
        return self.last_block['index'] + 1
    else:
        raise Exception("Invalid transaction")
    return transaction
```

06

Wallet



```python
def create_transaction(self, recipient: str, amount: float):
    transaction = {
        'sender': self.public_key,
        'recipient': recipient,
        'amount': amount,
    }

    # Sign the transaction
    transaction_hash = sha256(str(transaction).encode()).digest()
    signer = pkcs1_15.new(RSA.import_key(self.private_key))
```

```python
def update_utxos(self, transaction):
    # Remove UTXOs that were spent in the transaction
    self.utxos = [utxo for utxo in self.utxos if utxo not in transaction['inputs']]

    # Add new UTXOs that were created in the transaction
    self.utxos.extend(transaction['outputs'])
```

```python
def get_balance(self, *args):
    return sum(utxo['amount'] for utxo in self.utxos if not utxo['spent'])
```

```python
def public_key_to_address(self,public_key, *args):
    # Hash the public key using SHA256
    sha256 = hashlib.sha256()
    sha256.update(public_key)
    hashed_public_key = sha256.digest()

    # Encode the hashed public key in base58
    address = base58.b58encode(hashed_public_key)

    return address.decode()
```

```python
@app.route('/create_transaction', methods=['POST'])
def create_and_add_transaction():
    data = request.get_json()
    recipient = data.get('recipient')
    amount = data.get('amount')
    print(recipient,amount)
    if recipient is None or amount is None:
        return jsonify({'error': 'Missing recipient or amount'}), 400
    transaction = wallet.create_transaction(recipient, amount)
    blockchain.new_transaction(transaction, wallet.private_key)
    return jsonify({'transaction': transaction}), 200
```

Goal6: Wallet

Transaction Creation:

- Create transaction containing transaction's details and signature

UTXO Management and Balance Calculation:

- Update the UTXOs list and calculate the balance

Address Generation:

- Generating the readable wallet address by hashing the public key

Integration with the blockchain:

- Receive and create transaction, and add it in blockchain by method blockchain.new_transaction()

END