

# PROJET DE DÉVELOPPEMENT S2

## COMPRENDRE ET IMPLEMENTER LES MECANISMES DU BITCOIN

Encadrante : *DUPRAZ Elsa*

Réalisateurs : *DAHOU MANE Mehdi, HAFID Fayçal,  
HOUICHA Maroua, MAACHOU Marouane*

# SOMMAIRE

- La plateforme, vue par l'utilisateur
- Structure globale : classes implémentées
  - Classe User
  - Classe Bitcoin
  - Classe Transaction
- Structure globale : BlockChain
  - Constitution d'un block
  - Traitement d'une transaction
- Contraintes du cahier des charges

# PLATEFORME VUE PAR L'UTILISATEUR

A ▶

B ▶

C  
options

Une base de donnée  
existe et contient les  
identifiants et  
utilisateurs ainsi que  
leurs mots de passe

# PLATEFORME VUE PAR L'UTILISATEUR

A ▶

Une base de donnée existe et contient les identifiants et utilisateurs ainsi que leurs mots de passe

B ▶

On demande à l'utilisateur de s'identifier (3 essais pour l'ID et 3 essais pour le mot de passe)

C

options

# PLATEFORME VUE PAR L'UTILISATEUR

A ▶

Une base de donnée existe et contient les identifiants et utilisateurs ainsi que leurs mots de passe

B ▶

On demande à l'utilisateur de s'identifier (3 essais pour l'ID et 3 essais pour le mot de passe)

C

options

Un menu apparaît, l'utilisateur peut choisir de :

- > Consulter son solde
- > Voir l'historique des transactions
- > Effectuer une transaction
- > Se déconnecter



# STRUCTURE GLOBALE : CLASSES IMPEMENTEES

## USER



**UsersList** = [ ... ]  
 nom  
 prenom  
 ID  
 Wallet = [ ... ]  
 sk (clé privée)  
 Transactions = [ ... ]  
 pendingBitcoins = [ ... ]

## BITCOIN



**addressesList** = []  
**BitcoinList** = [ ... ]  
 nbObjects=0  
 maxValue = 20  
 currentValue  
 value  
 address  
 data = [ **state**, [ ... ] ]  
 signature

## TRANSACTION



**TransactionsList** = [ ... ]  
 sender   
 receiver   
 timestamp  
 amount  
 Inputs = [ ... ]  
 Outputs = [ ... ]  
 Block = [ ]

# STRUCTURE GLOBALE : CLASSES IMPEMENTEES

- Méthodes principales

**USER**



```
def getUserFromID(ID):  
    for user in User.UsersList :  
        if user.ID == ID :  
            return user  
    return None
```

```
def getSolde(self):  
    solde=0  
    for b in self.Wallet:  
        solde=solde+b.value  
    return solde
```

# STRUCTURE GLOBALE : CLASSES IMPEMENTEES

- Méthodes principales

Signature retournée par l'ECDSA :  
(R,S)

**USER**



```
def sign(self, bitcoin):  
    sk = SigningKey.generate(curve=ecdsa.SECP256k1)  
    signature=sk.sign(str(bitcoin).encode())  
    return (signature, sk.get_verifying_key())
```

Clé publique qui a créé la  
signature



# STRUCTURE GLOBALE : CLASSES IMPEMENTEES

- Méthodes principales

**USER**



Fonction d'ECDSA qui va recalculer R à partir de S et de la clé publique

Signature retournée par l'ECDSA :  
(R,S)

Clé publique qui a créé la signature

```
def sign(self, bitcoin):  
    sk = SigningKey.generate(curve=ecdsa.SECP256k1)  
    signature=sk.sign(str(bitcoin).encode())  
    return (signature, sk.get_verifying_key())  
  
def verify (self, bitcoin, signature):  
    allGood=True  
    currentMessage=str(bitcoin).encode()  
    vk = signature[1]  
    print("==> "+str(signature[1]) )  
    allGood=vk.verify(signature[0], currentMessage)  
    return allGood
```

# STRUCTURE GLOBALE : CLASSES IMPEMENTEES

## • Méthodes principales

**BITCOIN**



Vérifie si user possède le bitcoin

Donne le bitcoin à user

Tri du portefeuille

```
def getCurrentUser(self):
    return self.data[1][-1]
def addUser(self, user):
    self.data[1].append(user)
```

Retourne le possesseur du bitcoin

Rajoute l'utilisateur user comme possesseur actuel du bitcoin

```
def IsCurrentUser(self, user):
    return self.getCurrentUser()==user
def giveBitcoin(self, user):
    self.data[1].append(user)
    #user.Wallet.append(self)
    Bitcoin.putInRightPlace(self, user.Wallet)
```

```
def putInRightPlace(bitcoin, wallet):
    wallet.append(bitcoin)
    for i in range(len(wallet)) :
        for j in range(len(wallet)-1) :
            if wallet[j].value > wallet[j+1].value :
                tmp=wallet[j]
                wallet[j]=wallet[j+1]
                wallet[j+1]=tmp
```

# STRUCTURE GLOBALE : CLASSES IMPEMENTEES

- Méthodes principales

**BITCOIN**



Retourne la liste des users qui ont possédé le bitcoin, du premier possesseur jusqu'au plus récent

```
def trace(self):  
    if len(self.data[1]) > 1 :  
        return self.data[1][::-1]  
    return self.data[1]  
  
def previousOwners(self) :  
    display="Owners from most recent to least recent :\n\n"  
    liste=self.trace()  
    for element in liste :  
        display+=str(element)+"\n"  
    return display
```

Permet simplement d'afficher la liste

# STRUCTURE GLOBALE : CLASSES IMPEMENTEES

- Méthodes principales

**BITCOIN**



```
def ShatterBitcoin(self, serf1, serf2):  
    b1=Bitcoin(serf1)  
    b2=Bitcoin(serf2)  
    b1.data[1]=self.data[1]  
    b2.data[1]=self.data[1]  
    Bitcoin.addressesList.remove(self.address)  
    self.nbObjects+=1  
    self.BitcoinList.append(b1)  
    self.BitcoinList.append(b2)  
    user=self.getCurrentUser()  
    user.Wallet.remove(self)  
    user.Wallet.append(b1)  
    user.Wallet.append(b2)  
  
    return (b1,b2)
```

Valeur  
 $V=v1+v2$



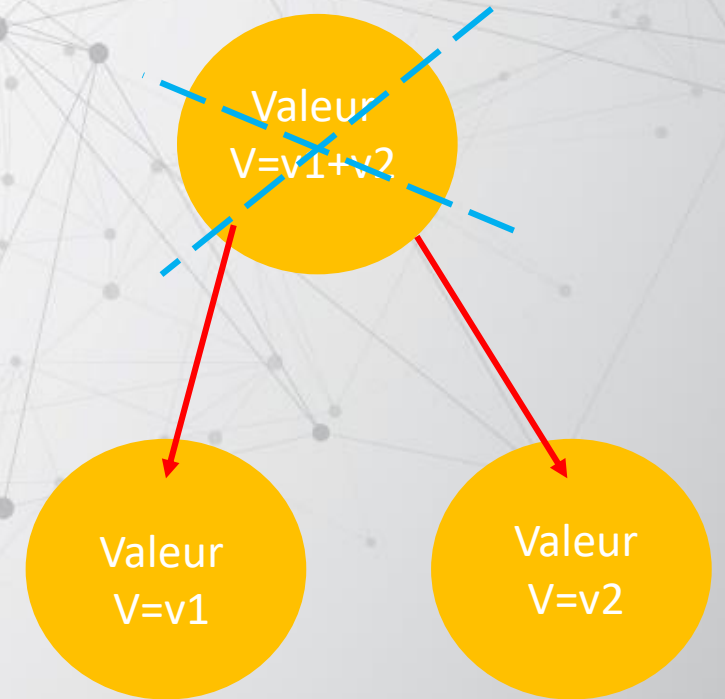
# STRUCTURE GLOBALE : CLASSES IMPEMENTEES

- Méthodes principales

**BITCOIN**



```
def ShatterBitcoin(self, serf1, serf2):  
    b1=Bitcoin(serf1)  
    b2=Bitcoin(serf2)  
    b1.data[1]=self.data[1]  
    b2.data[1]=self.data[1]  
    Bitcoin.addressesList.remove(self.address)  
    self.nbObjects+=1  
    self.BitcoinList.append(b1)  
    self.BitcoinList.append(b2)  
    user=self.getCurrentUser()  
    user.Wallet.remove(self)  
    user.Wallet.append(b1)  
    user.Wallet.append(b2)  
  
    return (b1,b2)
```





# STRUCTURE GLOBALE : CLASSES IMPEMENTEES

- Méthodes principales : Pas de méthode, mais traitement initial

## TRANSACTION



```
if sender.getSolde() < amount:
    print("solde insuffisant")
else :
    Transaction.TransactionsList.append(self)
    t=0
    out=0
    liste=self.sender.Wallet
    while not out :
        if liste[t].value < amount :
            t=t+1
        else :
            out=1
            (b1,b2)=liste[t].ShatterBitcoin(amount,liste[t].value-amount)
```

On vérifie que le sender possède un solde suffisant

On cherche, dans le portefeuille du sender, le bitcoin de valeur immédiatement supérieure à la valeur à envoyer afin de le monétiser

# STRUCTURE GLOBALE : CLASSES IMPEMENTEES

- Méthodes principales : Pas de méthode, mais traitement initial

## TRANSACTION



```

if b1.value == amount :
    b1.pending(sender, receiver, self)
    #Bitcoin.putInRightPlace(b2, sender.Wallet)
    self.inputs.append(b2)
    self.outputs.append(b1)
    #print("b1 is : "+str(b1.signature))
    print("Verification of the signature : \n"+str(receiver.verify(b1,b1.signature)))
else :
    b2.pending(sender, receiver, self)
    #Bitcoin.putInRightPlace(b1, sender.Wallet)
    self.inputs.append(b1)
    self.outputs.append(b2)
    print("Verification of the signature : \n"+str(receiver.verify(b1,b1.signature)))
    print("Transaction effectuée avec succès, attente de validation par la Blockchain :\n"+str(self))
    self.sender.Transactions.append(self)
    self.receiver.Transactions.append(self)
  
```

La méthode pending est une méthode de la classe Bitcoin qui agit sur les deux utilisateurs sender et receiver

Après monétisation: Le bitcoin de valeur=amount va dans les outputs (chez le receiver), et le second bitcoin généré reste dans les inputs (chez le sender)

# STRUCTURE GLOBALE : CLASSES IMPEMENTEES

- Méthodes principales : Pas de méthode, mais traitement initial

## TRANSACTION



Pendant la transaction, les bitcoins concernés n'appartiennent ni au sender ni au

```
def pending(self, sender, receiver, Transaction): receiver
    self.data[0]=1 # == en cours de transaction
    self.removeBitcoin(sender)
    if sender == receiver :
        pass
    else :
        sender.pendingBitcoins.append(["Out", self, Transaction])
        receiver.pendingBitcoins.append(["In", self, Transaction])
        self.signature=sender.sign(self)
```

Ils existent à la fois chez le sender et chez le receiver dans l'attribut pendingBitcoins, en mode « In » pour le receiver et en mode « Out » pour le sender

# STRUCTURE GLOBALE : CLASSES IMPEMENTEES

- Méthodes principales : Pas de méthode, mais traitement initial

## TRANSACTION



C'est à ce niveau qu'on fait signer chaque bitcoin par le sender

```
def pending(self, sender, receiver, Transaction):  
    self.data[0]=1 # == en cours de transaction  
    self.removeBitcoin(sender)  
    if sender == receiver :  
        pass  
    else :  
        sender.pendingBitcoins.append(["Out", self, Transaction])  
        receiver.pendingBitcoins.append(["In", self, Transaction])  
        self.signature=sender.sign(self)
```

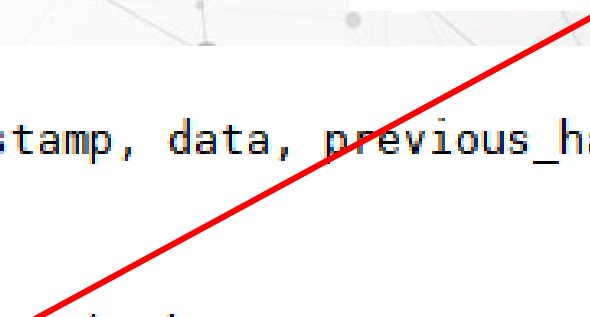


# STRUCTURE GLOBALE : CLASSES IMPEMENTEES

- Blockchain = Liste de blocks
- Contenu d'un bloc :

```
class Block:  
    def __init__(self, index, timestamp, data, previous_hash, proof):  
        self.index = index  
        self.timestamp = timestamp  
        self.data = data  
        self.previous_hash = previous_hash  
        self.hash = self.hash_block()  
        self.proof = proof
```

```
def hash_block(self):  
    sha = hasher.sha256()  
    block = str(self.index) + str(self.timestamp) + str(self.data) + str(  
        self.previous_hash  
    )  
    sha.update(block.encode('utf-8'))  
    return sha.hexdigest()
```





# STRUCTURE GLOBALE : CLASSES IMPEMENTEES

- Blockchain = Liste de blocks
- Contenu d'un bloc :

```
class Block:
    def __init__(self, index, timestamp, data, previous_hash, proof):
        self.index = index
        self.timestamp = timestamp
        self.data = data
        self.previous_hash = previous_hash
        self.hash = self.hash_block()
        self.proof = proof
```

- Prendre la chaine  
« last\_proof+data »
- Rajouter un nombre à la chaine
- Calculer le hash de la chaine
- Incrémenter le nombre et répéter  
jusqu'à ce que le hash produit  
commence par « 0000 »

## Preuve par travail

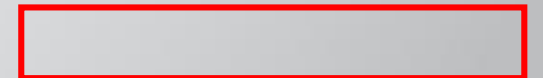
```
def proofOfWork2(last_proof, current_data) :
    origin=str(last_proof)+str(current_data)
    temp=0
    proved=origin+str(temp)
    sha=hasher.sha256()
    sha.update(proved.encode())
    sha=sha.hexdigest()
    string=str(sha)
    while string[:4] != "0000" :
        temp+=1
        proved=origin+str(temp)
        sha=hasher.sha256()
        sha.update(proved.encode())
        sha=sha.hexdigest()
        string=str(sha)
    print("temp="+str(temp))
    return temp
```

# STRUCTURE GLOBALE : CLASSES IMPEMENTEES

- Création d'un nouveau block (traitement d'une transaction)

Routine exécutée avant  
chaque traitement pour  
supprimer les éventuelles  
fraudes

```
def routineCheck():  
    print("\n\tRoutine check Ongoing.\n")  
    for user in User.UsersList :  
        for bitcoin in user.Wallet :  
            if bitcoin not in Bitcoin.BitcoinList :  
                input("\n\tForgery found and deleted :\nBitcoin of value "  
                user.Wallet.remove(bitcoin)  
                del bitcoin
```



# STRUCTURE GLOBALE : CLASSES IMPEMENTEES

- Création d'un nouveau block (traitement d'une transaction)

On commence par le receiver

```
def transactionTreatment(last_block):  
    Blocks=[last_block]  
    current=0  
    for transaction in Transaction.TransactionsList :  
        routineCheck()  
        print("Transaction en cours : "+str(transaction))  
        if not len(transaction.block) : #if it wasn't already treated  
            sender=transaction.sender  
            receiver=transaction.receiver  
            allGood=True  
            for pending in receiver.pendingBitcoins :  
                state = pending[0]  
                bitcoin = pending[1]  
                trans = pending[2]  
                if trans == transaction :  
                    allGood*=receiver.verify(bitcoin, bitcoin.signature)  
                    if allGood and state == "In" :  
                        bitcoin.data[0]=0  
                        bitcoin.giveBitcoin(receiver)  
                        bitcoin.removeBitcoin(sender)  
                        receiver.pendingBitcoins.remove(pending) #no longer
```

Pour chaque Bitcoin, on vérifie la validité de la signature

Si tout est bon, on rajoute le Bitcoin au receiver (on le met dans son portefeuille)

# STRUCTURE GLOBALE : CLASSES IMPEMENTEES

- Création d'un nouveau block (traitement d'une transaction)

On passe au sender

```
for pending in sender.pendingBitcoins :  
    state = pending[0]  
    bitcoin = pending[1]  
    trans = pending[2]  
    if trans == transaction and allGood :  
        if state == "Out" :  
            sender.pendingBitcoins.remove(pending)
```

Les signatures ayant déjà été vérifiées par le receiver, on retire directement les bitcoins du sender de son attribut pendingBitcoins



# STRUCTURE GLOBALE : CLASSES IMPEMENTEES

- Création d'un nouveau block (traitement d'une transaction)

```
if allGood :
```

```
    print("A new transaction has been verified.")
    data=str(transaction)
    newBlock=Block(Blocks[current].index+1,date.datetime.now(),
    data,Blocks[current].hash,proofOfWork(Blocks[current].proof))
    Blocks.append(newBlock)
    current+=1
    transaction.block.append(newBlock)
```

Si toutes les vérifications précédentes sont valides, on crée un nouveau block

Indique que la transaction a été traitée (pour ne pas la retraiter plusieurs fois)

Calcul de la preuve par travail (temps d'attente avant de rajouter le block à la blockchain)



# CONTRAINTES DU CAHIER DES CHARGES

L'intérêt de l'attribut Block de la transaction

Transparence

Permettre aux utilisateurs de consulter toutes les transactions effectuées et vérifier leur état

Tests par plusieurs cas possibles (cas de transaction en cours, transaction finie, cas de double-spending, ...)

Confidentialité

La confidentialité des utilisateurs doit être garantie

Simulations interactives entre plusieurs utilisateurs

L'intérêt de la liste pending Bitcoins

Usage d'une base de données SQL

Sécurité

Les transactions doivent être sécurisées (rétractabilité, fiabilité, modifications impossibles, ...)

Simulations de cas divers par essai de modifications et traçabilité d'une transaction arbitrairement choisie

- Routine de vérification avant chaque transaction par la Blockchain
- Les bitcoins sont limités par une valeur maximale, on ne peut en générer
- Possibilité de tracer un bitcoin