Projet Chablex - Manuel développeur

Table des matières

1. Introduction

Ce document vise à expliquer le fonctionnement des *Smart Contracts* et de la plateforme développés pour le projet Chablex. Il permet aussi d'expliquer l'environnement utilisé pour ce projet (encore en mode simulation). L'ensemble du projet est disponible sur le site https://github.com/jimy74/Chablex.

2. Environnement

2.1 *Smart Contracts*

Outils de développement des *Smart Contracts* : Ethereum Remix

Mise en route :

1. Aller sur le l'IDE en ligne pour développer des *Smart Contracts* : https://remix.ethereum.org/
2. Définir l'environnement JavaScript VM

2.2 Application coté client

Système d'exploitation : Ubuntu (sur Windows avec virtual box).

Framework : Embark 2.5.2

Outil de développement (IDE) : Atom

Navigateur utilisé pour les tests : Firefox

Mise en route :

1. Installer Embark

```
$ npm -g install embark
```

2. Installer le simulateur

```
$ npm -g install ethereumjs-testrpc
```

3. Créer un projet de démonstration

```
$ embark demo

$ cd embark_demo
```

4. Démarrer le simulateur

```
$ embark simulator
```

5. Lancer Embark

```
$ embark run
```

6. Ouvrir Atom

```
$ atom .
```

Mise à jour :

```
$ npm uninstall -g embark-framework

$ npm install -g embark
```

Puis procéder à la mise en route ci-dessus.

Mise en route d'Embark :

Iurimatias. (s.d.) *What is Embark*. Consulté le 25 Septembre 2017 à.:

https://github.com/iurimatias/embark-framework

2.3 Problèmes rencontrés

Problème : Gaz insuffisant pour que Embark compile les *Smart Contracts*

Solution : Augmenter le gaz limite dans le fichier

Dans le fichier embark_demo/config/contracts.json :

```
{
  "default": {
    "gas": "auto",
    "contracts": {
      "MonContract1": {"gas":3000000, "args":[10000]},
      "MonContract2": {"gas":4500000, "args":[10000]},
      "MonContract3": {"gas":3000000, "args":[10000]}
```

```
    }

  }

}
```

Problème : Embark indique qu'il ne trouve pas de *Blockchain*

Solution :

a) Arrêter Embark en pressant les touches `CTRL + C`

b) Arrêter le simulateur en pressant les touches `CTRL + C`

c) Taper la commande `embark blockchain` puis l'arrêter après qu'il initialise la *Blockchain* en pressant les touches `CTRL + C`

d) Puis relancer le simulateur avec la commande `embark simulator`

e) Dans une autre console, relancer Embark avec la commande `embark run`

Problème : Une transaction coûte trop cher en gaz et ne peut être exécutée

Solution : Augmenter la limite de gaz lors de l'appel de cette transaction

Par exemple si dans MonContract.sol on veux appeler maFonction prenant deux paramètres:

```
MonContract.maFonction([param_1],[param_2],{gas:4712388,gasPrice:10^11}).then(

    function(value) {

      console.log("Action accomplie :" + value);

    }).catch(function(error){

      console.log("Erreur :" + error);

    });
```

Problème : En mode simulation, on est le seul à faire des transactions sur la *Blockchain* locale, si on ne fait pas de transaction le temps reste figé (temps obtenu par l'instruction *now* dans un *Smart Contract*)

Solution : Ecrire un *timer* en JavaScript qui, tant que le temps n'a pas évolué depuis x secondes, va mettre à jour le temps localement (temps inchangé - x).

Problème : Comment tester l'application en simulant plusieurs comptes différents

Solution : Embark simulator permet d'utiliser 10 comptes de tests, ceux-ci sont reconnus automatiquement par web3. Il suffit ensuite de choisir l'un de ces comptes et de le passer en paramètre lorsqu'on appelle une fonction du *Smart Contract*.

```
var monCompte = web3.eth.accounts[0]; //Le premier des 10 comptes de simulations

MonContract.maFonction([param_1],[param_2],{from:monCompte}).then(

    function(value) {

      console.log("Action accomplie :" + value);

    }).catch(function(error){

      console.log("Erreur :" + error);

    });
```

Problème : Embark ne veut pas instancier un *Smart Contract* servant d'interface (par exemple ERC20.sol et ERC20Basic.sol de la librairie Open Zeppelin)

Solution : Réaliser cette interface dans vos *Smart Contracts* concrets et supprimer l'interface (par exemple dans BasicToken.sol et StandardToken.sol de Open Zeppelin)

3. Structure des Smart Contracts

4. Implantation de la logique de base et des standards

4.1 SafeMath

Eviter de vérifier les bugs d'*over flow* en utilisant, par exemple, à la place de l'opération "+" la méthode .add().

Les quatre opérations élémentaires sont présentes : add(); sub(); mul(); div().

Il suffit donc d'écrire en début de *Smart Contract* :

```
          using SafeMath for uint256;
```

Puis il sera possible de faire :

```
uint256 a = 1;

uint256 b = 3;

uint256 resultat = a.add(b); //Le résultat vaut 4
```

Ces fonctions sont implantées dans le *Smart Contract* "SafeMath.sol"

4.2 Rendre le *Token* transférable

Rendre le *Token* transférable en respectant le standard ERC-20

Le *Token* doit implémenter les fonctions suivantes :

```
// https://github.com/ethereum/EIPs/issues/20

contract ERC20Interface {

     // Get the total Token supply

     function totalSupply() constant returns (uint256 totalSupply);

     // Get the account balance of another account with address _owner

     function balanceOf(address _owner) constant returns (uint256 balance);

     // Send _value amount of tokens to address _to

     function transfer(address _to, uint256 _value) returns (bool success);

     // Send _value amount of tokens from address _from to address _to

     function transferFrom(address _from, address _to, uint256 _value) returns (bool
   success);
```

```
    /* Allow _spender to withdraw from your account, multiple times, up to the _value
  amount. */

    /* If this function is called again it overwrites the current allowance with
  _value. */

    // this function is required for some DEX functionality

    function approve(address _spender, uint256 _value) returns (bool success);

    // Returns the amount which _spender is still allowed to withdraw from _owner

    function allowance(address _owner, address _spender) constant returns (uint256
  remaining);

    // Triggered when tokens are transferred.

    event Transfer(address indexed _from, address indexed _to, uint256 _value);

    // Triggered whenever approve(address _spender, uint256 _value) is called.

    event Approval(address indexed _owner, address indexed _spender, uint256 _value);

}
```

TheEthereumWiki, *ERC20 Token Standard*, consulté le 21 Septembre 2017 à https://theethereum.wiki/w/index.php/ERC20_Token_Standard

Ces fonctions sont implantées dans les *Smart Contracts* "BasicToken.sol" et "StandardToken.sol".

De plus, le *Token* doit posséder les informations suivantes :

```
    string public constant symbol = "FIXED";

    string public constant name = "Example Fixed Supply Token";

    uint8 public constant decimals = 18;

    uint256 _totalSupply = 1000000;
```

TheEthereumWiki, *ERC20 Token Standard*, consulté le 21 Septembre 2017 à https://theethereum.wiki/w/index.php/ERC20_Token_Standard

Ces constantes sont définies dans le *Smart Contract* "ChabToken.sol".

5. Implantation de la logique métier

5.1 Retirer les droits de créateurs après 90 jours

L'adresse utilisée pour publier le contrat proviendra d'un porte-monnaie multi-signatures et dépendra donc de l'autorité de plusieurs personnes dont un huissier. Leurs privilèges, en tant que créateurs, par exemple l'ajout de membres et la distribution des *Tokens*, seront inaccessibles après la période de lancement de 90 jours (appelée Q1).

La période Q1 permet de définir correctement les membres de la communauté (de la monnaie locale) qui doivent correspondre à certains critères tels que: posséder une entreprise et appartenir à la localité de cette nouvelle crypto-monnaie. Cette période permet aussi de mettre en circulation les tout premiers *Tokens* vendus aux membres au prix de 1 CHF, ceux-ci seront mis en sécurité dans le coffre d'une banque Suisse locale.

Après la période Q1, ce sont les membres eux-mêmes qui pourront voter sur l'adhésion des nouveaux membres, mais aussi pourront commencer à transférer, demander ou faire des prêts.

Cette règle est implantée dans le *Smart Contract* "MomentalyOwned.sol" comme ceci :

```
contract MomentalyOwned is StandardToken {

     address public owner;

     uint public creationTime; /*Not a constant because "now" should not be initialize
    in compile-time */

     uint public constant periodQ1 = 90 days;

     function MomentalyOwned() {owner = msg.sender; creationTime = now;}

     modifier onlyOwner {require(msg.sender == owner);_;}

     modifier onlyInQ1 {require(now <= creationTime.add(periodQ1));_;}

     modifier onlyAfterQ1 {require(now > creationTime.add(periodQ1));_;}

    /* _; is used to indicate to the function modified where it should be inserted */

   //The continuation of the contract ...
```

source initiale : "Ownable.sol" de la librairie Open Zeppelin :

Open Zeppelin, (n.d.). *Zeppelin Solidity.* Repéré le 26 Septembre 2017 à https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/ownership/Ownable.sol

5.1.1.1.1 Inscription initiale des membres

Un membre est une structure (simple *record*) définie comme ci-dessous:

```
    struct Member {
        address member; //Ethereum public key of the member
        string name; //Official name of the member
        uint256 memberSince; //Date and time when the member is added
    }
```

La liste des membres est donc faite dans un simple tableau de membres comme ceci :

```
    Member[] public members;
```

Les créateurs du *Token* pourront ajouter des membres pendant la période initiale Q1.

Un membre se constitue d'un nom, identifiant l'entreprise locale, et de sa clé publique Ethereum.

Pour ajouter un membre il suffit donc d'appeler la fonction addMember :

```
function addMember(address targetMember, string memberName)
onlyOwner onlyNotMembers(targetMember) onlyInQ1 {
        uint256 id;
        memberId[targetMember] = members.length;
        id = members.length++;
        members[id] = Member({
      member: targetMember, memberSince: now, name: memberName});
        MembershipChanged(targetMember, true);
    }
```

En cas d'erreur de la part des créateurs, lors de la période Q1, ils pourront aussi supprimer un membre durant cette phase.

```
function removeMember(address targetMember)
onlyOwner onlyInQ1 onlyMembers(targetMember) returns (bool){
    for (uint256 i = memberId[targetMember]; i < members.length - 1; i++)
        members[i] = members[i+1];
    memberId[targetMember] = 0;
    balances[targetMember] = 0;
    delete members[members.length-1];
    members.length--;
    MembershipChanged(targetMember, false);
}
```

Il est possible de contrôler si une adresse fait bien partie des membres grâce aux deux *modifiers* suivants :

```
     //Modifier that allows only members
    modifier onlyMembers(address addr) {
        require(memberId[addr] != 0);
        _; //Indique où insérer le code de la fonction appelante
    }
    //Modifier that allows only not members
    modifier onlyNotMembers(address addr) {
        require(memberId[addr] == 0);
        _;
    }
```

Le *onlyMembers* est utilisé sur toute les fonctions liées aux *Tokens* et aux votations de nouveaux membres alors que *onlyNotMembers* est utilisée pour être sûr qu'on n'ajoute pas

deux fois la même adresse dans la liste des membres, car cela reviendrait à lui doubler le nombre de vote qu'il peut émettre pour l'élection de nouveaux membres.

Ces fonctions et *modifiers* sont définis dans le *Smart Contract* "CongressOwned.sol".

Source Initiale :

Vitiko. (n.p.) *Congress.* Consulté le 18 Août 2017 à

https://github.com/vitiko/solidity-test-example/blob/master/contracts/Congress.sol

5.2 Distribution des *Tokens*

Les créateurs peuvent distribuer des *tokens* aux membres seulement pendant la période Q1. Cette fonctionnalité permet aux créateurs de vendre des *Tokens* aux nouveaux membres, le prix initial étant défini à 1 CHF par *Token*.

Cette fonction est définie dans *le Smart Contract* "MintableToken.sol" (initialement de la librairie *Open Zeppelin*) de la manière suivante :

```
contract MintableToken is CongressOwned {

  event Mint(address indexed to, uint256 amount);

   /** @dev Function to mint tokens

    *  @param _to The address that will receive the minted tokens.

    *  @param _amount The amount of tokens to mint.

    *  @return A boolean that indicates if the operation was successful. */

  function mint(address _to, uint256 _amount) onlyInQ1 onlyOwner onlyMembers(_to){

    totalSupply = totalSupply.add(_amount);

    balances[_to] = balances[_to].add(_amount);

    Mint(_to, _amount);

    Transfer(0x0, _to, _amount); /* Detail : 0x0 Because we create new tokens in the
minting process (there is no sender of the transfer in term of loss)*/

  }
```

5.3 Crypto-monnaie pour des membres autogérés

Les actions possibles comme: le vote de nouveaux membres, le transfert, mais aussi la demande de prêt sont accessibles uniquement par des membres.

Il sera possible, pour les membres après la période Q1, de proposer de nouveaux membres et de voter pour ceux qui sembles de confiance (car faisant partie de la localité ou d'une région proche). Une proposition de nouveau membre est retenue si elle a reçu au moins 67% de votes positifs avant la fin du temps imparti (disons un mois). Ces paramètres sont des constantes du *Smart Contract* "CongressOwned.sol" , ils ne peuvent être définis, uniquement, avant la compilation et donc forcément avant sa publication sur la *Blockchain* Ethereum.

La proposition d'un nouveau membre est implantée dans le *Smart Contract* "CongressOwned.sol" comme suit :

```
function newProposal( address candidateAddress,string candidateName)
onlyAfterQ1 onlyMembers(msg.sender) onlyNotMembers(candidateAddress)
 returns (uint256 proposalID) {
    //Sender did not make a proposal for a while
    require(now >= timeLastProposal[msg.sender] + periodEnterProposal);
    //Update the time of his last proposal from the sender
    timeLastProposal[msg.sender] = now;
    //Create a new proposal :
    //Set the id of the new proposal and (after) increase the proposals array
    proposalID = proposals.length++;
    Proposal storage p = proposals[proposalID]; //Set the pointer
    p.id = proposalID; //Set the id of this proposal
    p.candidateAddress = candidateAddress; //Set the candidate ETH address
    p.candidateName = candidateName; //Set the candidate firm identifier
    p.votingDeadline = now + debatingPeriod; //Set the deadline of this proposal
    p.executed = false; //Set the proposal to unexecuted
    p.proposalPassed = false; /*Set the result of the proposal to false (unused if
    not executed) */
    //Vote for my own proposal :
    Vote storage v = p.votes[p.votes.length++]; //Get a new vote structure
```

```
    v.voter = msg.sender; //Set the voter

    v.inSupport = true; //Set the stat of his vote (accepted or rejected)

    v.justification = "Creator's vote"; //Set the justification

    p.voted[msg.sender] = true; //Sender has voted for this proposal

    p.numberOfVotes = 1; //Set the count of votes

    p.currentResult = 1; //Set the count of acceptations

    numProposals = proposalID +1; //Update the number of proposals

    ProposalAdded(proposalID, candidateAddress, candidateName);

    return proposalID;

    }
```

Les membres doivent être avertis que, une fois qu'un nouveau membre est accepté par votation, il n'est plus possible de le renier de la liste des membres. Ceci est bien entendu pour garantir la diversité et éviter les renversements de pouvoir où des membres bienveillants seraient supprimés par de nouveaux membres malveillants.

La possibilité de voter (d'accepter ou de rejeter) une proposition de nouveau membre est implantée dans le même *Smart Contract* comme ceci:

```
function vote(uint256 proposalID, bool supportsProposal, string justificationText)
onlyAfterQ1 onlyMembers(msg.sender) returns (uint256 voteID) {

    Proposal storage p = proposals[proposalID]; //Get the proposal

    require(p.voted[msg.sender] == false); //If has already voted, cancel

    Vote storage v = p.votes[p.votes.length++]; //Get a new vote structure

    v.voter = msg.sender; //Set the voter

    v.inSupport = supportsProposal; //Set the vote stat (accepted or rejected)

    v.justification = justificationText; //Set the justification

    p.voted[msg.sender] = true; //Set this voter as having voted

    p.numberOfVotes++; //Increase the number of votes

    if (supportsProposal) //If they support the proposal
```

```
                p.currentResult++; //Increase score
    // Fire Events
    ProposalTallied(proposalID,p.currentResult,p.numberOfVotes,p.proposalPassed;
}
function executeProposal(uint256 proposalID) onlyAfterQ1 {
    Proposal storage p = proposals[proposalID];
    require(now >= p.votingDeadline); //Has the voting deadline arrived?
    require(!p.executed); //Has it not been already executed
    require(p.numberOfVotes >= minimumQuorum); //Has a minimum quorum?
    //If difference between support and opposition is larger than margin
    if ( p.currentResult * 100 / p.numberOfVotes >= majorityMinPourcent) {
                //Add the member
                addElectedMember(p.candidateAddress,p.candidateName);
                p.proposalPassed = true;
    } else {
                p.proposalPassed = false;
    }
    p.executed = true; //Note the proposal as executed
    //Fire this event
    Voted(proposalID, supportsProposal, msg.sender, justificationText);
    return p.numberOfVotes;
}
```

Il suffit donc à quiconque sur le réseau Ethereum de demander l'exécution d'une proposition pour que cette proposition puisse être conclue sur une acceptation ou non du membre proposé. Une proposition ne peut être exécutée que si elle existe depuis plus d'une semaine et qu'elle a reçu au moins 3 votes (défini dans les constantes du *Smart Contract* "CongressOwned.sol").

5.4 Autoriser une demande de prêt

Le maximum empruntable est le nombre de *Tokens* maximum qu'un membre peut emprunter. Si le maximum empruntable est atteint, c'est-à-dire qu'il a emprunté jusqu'à atteindre son maximum empruntable (initialement fixé à 500 *Tokens*), il ne peut, dès lors, plus faire de demande à moins de rembourser, par la fonction de prêt, la totalité de ses emprunts à la communauté.

Une règle nommée "peutDemander" est implantée en Solidity en utilisant un *modifier*.

Celle-ci stipule que l'on peut demander une valeur X si ces 4 conditions sont réunies :

- Le demandeur a remboursé ses emprunts
- La valeur X demandée est supérieure à 1 *Token*
- On ne peut pas emprunter plus de Tokens que le maximum empruntable
- Le total des demandes + la demande ne dépasse pas le total des remboursements + le max
- Le total demandé par la communauté ne dépassera pas le 1/3 du nombre total de *Tokens*

Mais attention, d'autres règles, telles que "*onlyMember"* et "*onlyAfterQ1",* sont aussi là pour protéger contre une demande incongrue.

La règle "peutDemander" est implantée sous forme d'un *modifier* dans le *Smart Contract* "ChabToken.sol" de la manière suivante :

```
modifier peutDemander(uint256 _value) {

    require(_value >= 1); // La valeur demandée est au moins 1 token

    //Définir le maximum empruntable ou sa valeur initiale

    uint monMaxEmpruntable = getMaxEmpruntable(msg.sender);

    //N'emprunte pas plus de tokens que le max

    require(_value <= monMaxEmpruntable);

    /* Le total des demandes plus la demande, ne dépassent pas le total des
    remboursements plus le maximum empruntable */
```

```
        require(demandes[msg.sender].add(_value)<=
        remboursements[msg.sender].add(monMaxEmpruntable));

        //Le total demandé est inférieur à 1/ratio du nombre de tokens total

        require((_value + demandesEnCours.getTotalValue()) *
        minRatioCirculent < totalSupply);

        _; // Indique où insérer le code de la fonction appelante

    }
```

Remarquez que ce *Smart Contract* est codé et commenté en français car il a été conçu à partir de rien, alors que les autres contracts proviennent tous de librairies ou d'autres ressources internes puis ont étés modifiés. Le *Smart Contract* "ChabToken.sol" est donc le cœur métier du projet Chablex.

5.5 Demander un prêt

Pour tenter d'obtenir un prêt, il faut être autorisé, comme vu dans le chapitre ci-dessus, de plus, comme dit précédemment, il faut être membre et que la période Q1 soit terminée.

La fonctionnalité d'emprunt est implémentée dans le *Smart Contract* "ChabToken.sol" de la façon suivante :

```
        function demander(uint256 _value) public

        onlyAfterQ1 onlyMembers(msg.sender) peutDemander(_value) {

                    //Augmente le total demandé

                    demandes[msg.sender] = demandes[msg.sender].add(_value);

                    //Ajoute la demande à la file d'attente

                    demandesEnCours.addRequest(msg.sender, _value);

                    //Déclenche l'évènement

                    Demander(msg.sender, _value);

        }
```

On observe, au passage, que le code devient plus court grâce aux différents *modifiers* tels que *"onlyAfterQ1", "onlyMembers" et "peutDemander".*

5.6 Autoriser un prêt

On peut prêter la valeur X si ces trois conditions sont réunies :

- La valeur X prêtée est supérieure à 0 *Token*
- Le prêteur possède suffisamment
- La valeur X du prêt ne dépasse pas le total des demandes en cours

Cette règle est implantée dans le *Smart Contract* "ChabToken.sol" comme ceci :

```
    modifier peutPreter(uint256 _value) {
     // La valeur prêtée est supérieure à 0 token
     require(_value > 0);
     // Le prêteur possède suffisamment
     require(balances[msg.sender] >= _value);
                /* La valeur du prêt ne dépasse pas le total des demandes en cours,
           et les demandes complétées n'appartiennent pas au prêteur */
     require(demandesEnCours.containMinValueFromOther(_value, msg.sender));
     _;
    }
```

5.7 Emettre un prêt

Il est possible, pour un membre après la période Q1, de faire un prêt à la communauté, cela sera comptabilisé comme un remboursement de sa part et induira, peut-être (conditions expliquées dans le chapitre suivant), à une récompense (en augmentant son maximum empruntable).

La fonction "pêter" est définie dans le *Smart Contract* "ChabToken.sol" comme suit :

```
function preter(uint256 _value) public
onlyAfterQ1 onlyMembers(msg.sender) peutPreter(_value){
    // Déduit en amont la balance du prêteur
    balances[msg.sender] = balances[msg.sender].sub(_value);
```

```
// Initialise le montant qu'il reste à prêter
uint256 pretRestant = _value;
while (demandesEnCours.queueLength() > 0 && pretRestant > 0){
          /* Récupère la demande la plus vieille sans la dépiler
          (demandeur,valeur) */
          var (demandeur,valeur) = (demandesEnCours.copyPopRequest());
          // Si le prêt restant peut recouvrir cette demande intégralement
          if (pretRestant >= valeur){
                    demandesEnCours.popRequest();
                    //Augmente les emprunts total du demandeur
                    emprunts[demandeur] = emprunts[demandeur].add(valeur);

                    //Augmente les remboursements total du prêteur
                    remboursements[msg.sender] =
                    remboursements[msg.sender].add(valeur);
                    // Réduire la somme encore prêtable
                    pretRestant = pretRestant.sub(valeur);
                    Preter(demandeur,valeur);
          } else { //Sinon, répondre partiellement à cette demande
                    emprunts[demandeur]=emprunts[demandeur].add(pretRestant;
                    balances[demandeur]=balances[demandeur].add(pretRestant;
                    remboursements[msg.sender] =
                    remboursements[msg.sender].add(pretRestant);
                    /* Réduit la valeur de cette demande mais la laisse dans
                    la file d'attente */
                    demandesEnCours.replaceInFrontRequest(demandeur,
                    valeur.sub(pretRestant));
                    PreterUnePartie(demandeur, pretRestant);
```

```
                    // Arrête la boucle car tout prêté
                    pretRestant = 0;
                }
    }
    /* Et ici, donner selon certaines conditions, la récompense
       ce qui est expliqué dans le chapitre suivant ... */
}
```

5.8 Récompenser les prêteurs

Si le prêteur, une fois le prêt effectué correctement, se trouve dans la situation où il y a, en tout, prêté autant qu'il a emprunté, qu'il a déjà demandé jusqu'à sa limite maximale et que celle-ci n'a pas changé depuis 30 jours, alors le maximum qu'il peut emprunter la prochaine fois double.

Cette règle se trouve à la fin de la fonction de prêt dans le *Smart Contract* "ChabToken.sol", et est implantée comme ceci :

```
if (//Si le prêteur a remboursé ses emprunts
 remboursements[msg.sender] >= emprunts[msg.sender] &&
 //Et est arrivé à son maximum
 remboursements[msg.sender] >= getMaxEmpruntable(msg.sender) &&
 //Et que sa dernière augmentation date de au moins 30 jours
 now >= dateChangementMax[msg.sender].add(tempsMinChangeMax)
) {
    dateChangementMax[msg.sender] = now;
    //Multiplie (par 2) le maximum empruntable lors du prochain emprunt
    maxEmpruntable[msg.sender]                                          =
    getMaxEmpruntable(msg.sender).mul(facteurChangeMax);
    /* A noter que dans ce cas il peut aussi faire un nouvel emprunt
    (voir la première ligne du modifier peutDemander) */
}
```

5.9 Prêts automatiquement accordés aux demandeurs

Les prêts sont accordés grâce à une pile (First In First Out) qui est codée dans le *Smart Contract* "QueueDemande.sol". C'est un outil technique, il n'est donc pas nécessaire de l'analyser dans les détails pour comprendre le fonctionnement ChabToken. Cependant, la queue de demandes reste un point névralgique du système, elle reste une partie sensible qui mériterait d'être plus amplement revue et testée pour garantir la sécurité du système.

Cette fonctionnalité aurait besoin d'être améliorée, en tout cas pour vérifier que la pile (qui est en l'occurrence circulaire) ne finisse par réécrire sur elle même. Ceci n'est pas encore réalisé car le projet arrive à son échéance, une idée serait de tester la taille de la queue avant d'ajouter une nouvelle demande.

Vous retrouverez le *Smart Contract* "QueueDemande.sol" et tous les autres sur le GitHub du projet Chablex.

6. Plateforme Chablex

La plateforme Chablex est encore simulée dans un environnement de test en utilisant le *framework Embark* (voir la mise en route dans le chapitre Environnement).

Celui-ci simule une *Blockchain* localement et permet de manipuler plusieurs comptes et d'utiliser de faux Ethers afin d'éviter de payer réellement les transactions.

Les *Smart Contracts* doivent être mis dans le dossier des contracts, Embark va les compiler et les rendre accessibles en passant par du JavaScript.

Embark permet aussi de faire un nouveau projet de démonstration ou figure un exemple de code pour interagir avec un *Smart Contract* basic. Pour cela, il suffit de taper la commande "embark demo".

Cette plateforme respecte la structure d'un projet généré par Embark.

Cette plateforme coté client est donc codée en utilisant les langages bien connus des développeurs Web tel que HTML, CSS, et JavaScript (ainsi que jQuery et AJAX).

Pour mieux comprendre les rouages de cette plateforme, il est essentiel d'avoir compris cette structure (décrite ci-dessous).

6.1 Structure du projet

1. Dossier ou se trouvent les Smart Contracts à utiliser

2. Fichier CSS pour changer le style de la plateforme

3. Fichier JavaScript où on interagit avec les *Smart Contracts*

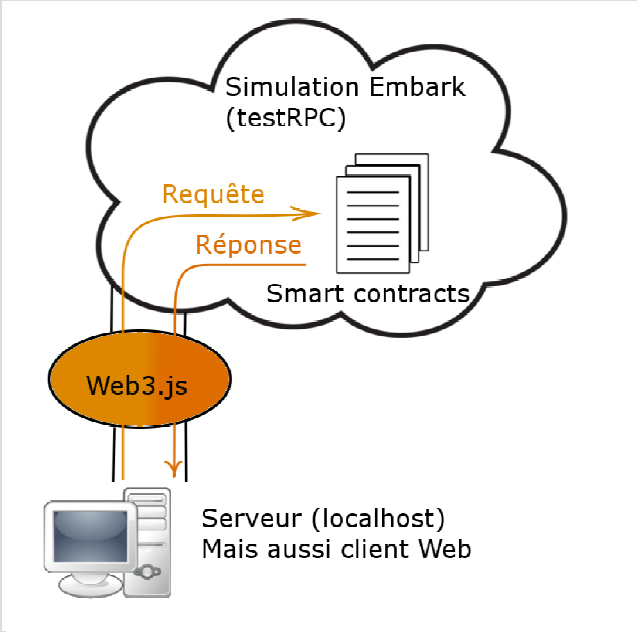
4. Ficher HTML contenant la structure de la plateforme

5. Ficher JSON permettant de modifier le gaz à utiliser pour compiler les Smart Contracts

6.2 Interactions avec les Smart Contracts (concept)

Les *Smart Contracts* sont pour, l'instant, publiés par Embark, dans une *Blockchain* de simulation.

Pour appeler une fonction d'un *Smart Contract*, le serveur (aussi simulé en local), va utiliser Web3.js pour effectuer des transactions sur la *Blockchain*. Une fois une fonction appelée (en JavaScript), Web3.js va ouvrir ce qui s'appelle une "*promise*", et vous transmettra de manière asynchrone le résultat obtenu suite à votre requête.



source de l'image : montage par mois même

Si une erreur imprévue dans le *Smart Contract*, telle que la rencontre de l'instruction "*throw*", le résultat n'est pas reçu, mais une exception est déclenchée.

6.3 Interactions avec les Smart Contracts (code)

Voici un exemple de code qui permet de visualiser comment interagir avec un *Smart Contract* (dans un projet Embark) :

```
MonContract.maFonction([param_1],[param_2]).then(

    function(value) {

      console.log("Action accomplie :" + value);

    }).catch(function(error){

      console.log("Erreur :" + error);

    });
```

7. Améliorations possibles

7.1 Côté *Smart Contract*

a) Utiliser des smart de gestion de cycle de vie

La librairie Open Zeppelin propose trois *Smart Contracts* réutilisables (Pausable.sol, Migration.sol et Destructible.sol). Ils permettraient aux créateurs, ou aux membres par votation (après ajout de quelques lignes de code), de stopper le *Smart Contract* en cas d'attaque afin d'éviter une catastrophe.

Cela faciliterait la légalisation du projet Chablex car il offrirait un moyen d'arrêter si le gouvernement l'ordonnerait, réduisant le risque d'interdiction d'un tel projet.

Mais cela permettrait surtout de mettre à jour le *Smart Contract* ChabToken, car bien qu'on ne le prévoie pas forcément dès le départ, cela peut devenir très vite une nécessité en cas de bug ou d'attaque.

Tout ceci reste à faire, pour l'instant le *Smart Contract* ChabToken n'offre, après la période initiale, aucun moyen de l'arrêter, mais aussi aucun moyen de le mettre à jour (de manière transparente pour l'application coté client).

b) Publier le *Smart Contract* sur un testNet Officiel

Cela permettrait d'accéder plus facilement au *Smart Contract* pour le tester, mais permettrait également de tester l'application coté client en utilisant cette fois MetaMask (un *wallet* d'Ether pour le navigateur Chrome).

c) Proposer des récompenses pour les chasseurs de bugs

Ce serait le début d'un *Smart Contract* sécurisé, malheureusement cette étape est coûteuse et nécessiterait un moyen de financement qui resterait à définir.

Une solution serait d'utiliser les Francs Suisses acquis pendant la vente initiale.

d) Tester le *Smart Contract* étape par étape

Ceci permettrait de, petit à petit, monter en exigence en réduisant le nombre de tests et donc le risque de failles de sécurité.

e) Publier la version finale du *Smart Contract* sur la *Blockchain* Ethereum

Cela se ferait sûrement sur le testNet car, pour l'instant, on n'utilise pas d'Ether dans les *Smart Contracts* (ce qui réduit les frais de transaction). Une autre solution serait

de le publier sur le mainNet qui a, au moins, l'avantage d'être sécurisé par l'existence d'un grand nombre de mineurs.

7.2 Côté application

a) Trier les tableaux qui sont désordonnés à cause de l'asyncronicité des requêtes.
b) Afficher le nombre de *Tokens* en circulation.
c) Afficher le temps restant pour une proposition avant quelle puisse être exécutée.
d) Effectuer des messages d'alertes plus fines.
e) Créer un graphique pour visualiser l'évolution de l'état d'un compte dans le temps.
f) Changer le code JavaScript pour que Web3.js se connecte au *Smart Contract* définitivement publié sur la *Blockchain*.
g) Changer le code JavaScript pour que web3.js afin qu'il se lie à MetaMask.

7.3 Créer un plan marketing et financier

a) Créer un site officiel
b) Mettre en place des moyens de communication
c) Rencontrer les entreprises locales
d) Trouver des moyens de financement

7.4 Recruter une équipe de développeurs aguerris

Cela serait une des meilleure garantie pour la réussite du projet, autant sur le plan technique que marketing. Il faudrait qu'une telle équipe puisse être rémunérée dans le cadre de ce projet (à moins qu'ils acceptent de travailler gratuitement, ce qui est tout de même peu probable). Une deuxième solution, plus économique, mais avec malheureusement moins de garantie, serait qu'un autre étudiant développe une version 2 du projet Chablex dans le cadre futur d'un travail de Bachelor.