**Notes sur le smart contract ADMM.sol**

1. Pour l’instant, on considère qu’un compte devra appeler les fonctions d’agrégation. Plus précisément :

* Il définit l’intervalle de temps du prochain créneau d’optimisation :

**function** setTimeInterval(**uint** \_startTime, **uint** \_endTime, **uint** \_timeStep) **public** onlyOwner {

// chose carefully startTime, endTime and timeStep

require(\_startTime < \_endTime, "startTime shouble be less than endTime");

require(\_timeStep < \_endTime - \_startTime, "impossible timestep");

startTime = \_startTime;

endTime = \_endTime;

timeStep = \_timeStep;

nbSteps = (\_endTime - \_startTime)/\_timeStep+1;

**for** (**uint** i=0; i < nbSteps; i++){

X\_global.push(0);

}

}

* Il définit ensuite la fonction de prix pour le prochain intervalle (remarque : est-ce que c’est vraiment à l’agrégateur de l’importe ? chaque utilisateur pourrait l’importer localement)

**function** setPrice (**uint**[] **memory** \_priceProfile) **public** onlyOwner {

// contractCreator can call this function to load the price vector for the desired period into the blockchain

require(\_priceProfile.length == nbSteps, "vector doesn't have good size");

**for** (**uint** i=0; i < nbSteps; i++){

priceProfile.push(\_priceProfile[i]);

}

}

1. **Pour les utilisateurs :**

* On définit une structure User, qui sauvera les informations utiles pour chaque utilisateur :

**struct** User {

**uint** userID;

**uint** class; // class (1:EV, 2: storage, 3: PV; 4: loads)

**int**[] optimProfile; // power profile of the user

//uint amount; // amount of money of the user ?

//int[] forecastProfile; // EST-CE UTILE ?

}

* On définit les mappings suivants :

**mapping**(**address** => User) **public** users;

**mapping**(**address** => **uint**) **public** registeredUsers; // (0: not registered, 1: already registered)

**mapping**(**address** => **uint**) **public** profileSend; // (0: profile not send, 1: already send)

Le premier associe une adresse Ethereum à une structure User. Chaque utilisateur est donc associé à une structure User par son adresse Ethereum.

Le deuxième mapping associe l’adresse Ethereum à un entier (0 ou 1) pour savoir si l’utilisateur a déjà créé une structure User (1) ou pas (0). Cela évite à un compte Ethereum de pouvoir créer plusieurs structures User.

De même, le troisième mapping est utilisé pour s’assurer que chaque utilisateur envoie une seule fois son profil de conso/ prod.

* On définit le tableau suivant :

**address**[] **public** userAccounts; // array that saves the address of users

Ce tableau stocke les addresses Ethereum des comptes qui ont créé une structure User.

* Pour créer une nouvelle structure User, chaque utilisateur doit appeler la fonction suivante :

**function** newUser(**uint** \_class) **public** **returns** (**uint** \_userID){

// user calls this function to declare its class

// each user gets a different \_userID

require(registeredUsers[msg.sender]==0);

\_userID = numUsers++;

**int**[] **memory** \_zero;

users[msg.sender] = User(\_userID,\_class, \_zero);

registeredUsers[msg.sender] =1; // prevent user to create twice a user struct

userAccounts.push(msg.sender);

}

Cette fonction vérifie que le compte Ethereum qui l’appelle n’a pas déjà créé une structure User. Ensuite, il associe un entier userID à chaque utilisateur.

* Les utilisateurs prennent connaissance de la function de prix en appelant la fonction : getPrice()
* Les utilisateurs appellent ensuite la fonction suivante pour envoyer leur profil de consommation ou production optimisé selon leurs préférences :

**function** sendOptimizedProfile(**int**[] **memory** \_optimProfile) **public** {

require(\_optimProfile.length == nbSteps, "vector doesn't have good size");

require(profileSend[msg.sender]==0); // check that the user has not already send the profile

User **storage** user = users[msg.sender];

user.optimProfile = \_optimProfile;

**for** (**uint** i = 0; i<nbSteps; i++) {

**if** (user.class == 3) {

user.optimProfile[i] = -\_optimProfile[i]; // negative values for PV generators

}

X\_global[i] += user.optimProfile[i]; // update sum of profiles

}

// send an event to inform that user has updated his profile

**emit** SendProfile(msg.sender);

}

A chaque appel de cette fonction, le vecteur X\_global qui somme les puissances de chaque profil à chaque instant est mis-à-jour. Le deuxième require permet de s’assurer que chaque utilistauer ne peut appeler qu’une seule fois cette fonction.

1. L’agrégateur compte combien d’utilisateurs ont envoyer leur profil (dans le script python).

Quand ce nombre est égal au nombre de joueurs, il appelle la fonction d’agrégation pour procéder à l’optimisation globale.