Laboratoire de PCO #3
Gestion d'accès concurrents

Rémi Ançay & Lucas Charbonnier

Introduction

Ce rapport contient le détail de la réalisation du labo 3 de PCO. Vous y trouverez notamment :

- Le détail des différentes modifications que nous avons apportées au code afin de faire fonctionner la simulation
- Les explications relatives à nos choix de conception
- La manière dont nous avons testé notre implémentation
- Les limites de notre solution

Implémentation

Ajout d'une mécanique de transaction

Après analyse, nous avons identifié les zones critiques du programme comme étant celles relatives aux échanges entre les différents acteurs. Afin que la simulation fonctionne en multithreading, il a donc fallu implémenter un système de transaction permettant aux différents acteurs de bloquer l'accès à leurs ressources pendant un moment.

Pour ce faire, nous avons implémenté deux méthodes dans la classe Seller :

```
    /**
     * @brief Indique qu'une transaction est en cours et que personne
d'autre ne doit accéder aux resources
     * Si une transaction est déjà en cours, cette méthode attendra la fin
de l'autre transaction avant de se terminer.
     */
    void startTransaction();

    /**
     * @brief Indique que la transaction est terminée
     */
    void finishTransaction();
```

Ces méthodes ont une visibilité « protected » et peuvent donc être appelées par les classes dérivées, càd, les grossistes, les usines et les extracteurs. L'utilisation des deux méthodes transactionnelles est très simple, il suffit d'appeler startTransaction() au début de la zone critique et stopTransaction() après la zone critique. Voici un exemple d'utilisation (tiré de la méthode buyResources() de Wholesaler) :

```
    startTransaction();

    bool transactionSuccessful = false;
    // si on a assez d'argent et que le vendeur peut nous vendre l'objet...
    if(money >= price) {
        if(seller->trade(itemToBuy, qty) != NO_TRADE) {
            // on effectue la transaction
```

```
            money -= price;
            stocks[itemToBuy] += qty;
            transactionSuccessful = true;
        }
    }

    finishTransaction();
```

L'implémentation de ces deux méthodes est très simple, startTransaction() lock un mutex et finishTransaction le libère. Ici, nous avons choisi d'encapsuler ce détail pour cacher l'utilisation du mutex à l'utilisateur de la classe Seller.

Pour garantir la protection des stocks et de l'argent de chaque Seller, il suffit donc d'englober toute modification de ces attributs dans une transaction. Nous avons donc modifié tous les endroits nécessaires, c'est-à-dire dans les méthodes suivantes :

- *Extractor::run()*
- *Factory::buildItem()*
- *Factory::orderResources()*
- *Seller::trade()*
- *Wholesaler::buyResources()*

Implémentation de trade() dans Seller

Lors de l'implémentation de la méthode trade() dans les classes dérivées telles que Factory ou Extractor, nous nous sommes rendus compte que l'implémentation de cette méthode serait identique dans toutes les classes dérivées. Pourtant, cette méthode était marquée comme virtuelle **pure** dans Seller.

Pour éviter d'avoir à implémenter plein de fois la même logique, nous avons laissé trade() comme étant virtuelle (mais pas pure) dans Seller et l'avons implémenté de la manière suivante :

```
int Seller::trade(ItemType what, int qty) {
    auto itemsForSale = getItemsForSale();

    // L'objet demandé est-il à vendre ?
    if(itemsForSale.find(what) == itemsForSale.end())
        return NO_TRADE;

    // L'objet demeandé est-il disponible ?
    if(stocks.find(what) == stocks.end())
        return NO_TRADE;

    startTransaction();

    // L'objet demandé est-il disponible dans la bonne quantité ?
    if(stocks[what] >= qty){
        // transaction
        int totalPrice = getCostPerUnit(what) * qty;
        stocks[what] -= qty;
        money += totalPrice;

        finishTransaction();
```

```
        return totalPrice;
    }

    finishTransaction();
    return NO_TRADE;
}
```

Vous noterez l'utilisation de la constante **NO_TRADE**. C'est une constante que nous avons ajouté qui vaut 0 et qui permet de mieux comprendre la signification de la valeur de retour.

Terminaison du programme

Pour terminer le programme, nous avons simplement ajouté une méthode requestStop() dans la classe Seller. Cette méthode va mettre un booléen à 1 et ce booléen se retrouve dans la condition de la boucle principale dans run().

Pour terminer le programme, on ne fait qu'appeler requestStop() sur tous les sellers :

```
void Utils::endService() {
    for(auto& e : extractors)
        e->requestStop();
    for(auto& f : factories)
        f->requestStop();
    for(auto& ws : wholesalers)
        ws->requestStop();

    std::cout << "It's time to end !" << std::endl;
}
```

Finalement, il ne reste plus qu'à joindre tous les threads et le programme pourra s'arrêter correctement.

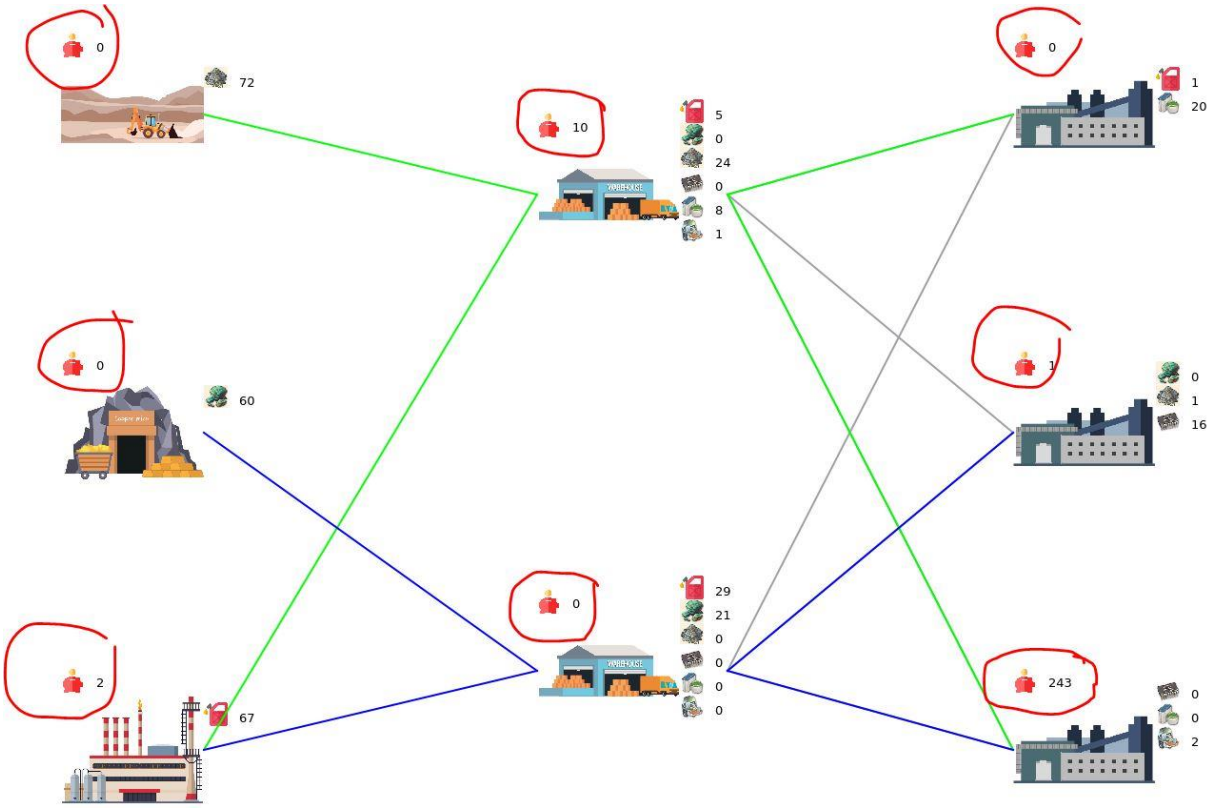
Tests et limites de notre implémentation

Notre solution a été testé principalement à la main. Nous n'avons pas écrit de tests unitaires car la grande majorité du code nous était déjà fournie et il n'était pas demandé de tester le code déjà existant. Le code que nous avons écrit était surtout relatif à l'implémentation du multithreading et aurait donc été très compliqué à tester.

Nous avons vérifié que les fonds de départ étaient équivalents aux fonds à la fin du programme. Nous avons également laissé tourner la simulation suffisamment longtemps pour s'assurer qu'il n'y avait pas de « deadlocks ». Ces deux vérifications, ainsi qu'un peu d'observation de l'interface, nous permettent de conclure que le multithreading est fonctionnel.

Limites de la simulation

Après avoir obtenu une simulation fonctionnelle, nous avons remarqué que la simulation finit toujours par se bloquer dans un état similaire à l'état suivant :



On peut voir que :

- Les grossistes ont épuisé tout leur budget
- Les usines sont:
    - Soit riches mais les grossistes n'ont pas les ressources dont elles ont besoin
    - Soit pauvres donc plus les moyens de se procurer les matières premières ou de payer leurs employés
- Les extracteurs n'ont plus de quoi payer leurs employés pour continuer à extraire des ressources.

Nous pensons que cela est dû au fait que les grossistes n'achètent pas de manière « intelligente » leurs ressources. Ils achètent des trucs au hasard en espérant que ça leur permette d'effectuer d'autres échanges sans prendre en compte le fait qu'ils n'ont pas une quantité illimitée d'argent. Pour résoudre ce problème, il faudrait repenser la manière dont les grossistes investissent leur argent mais cela dépasse le cadre de ce laboratoire.