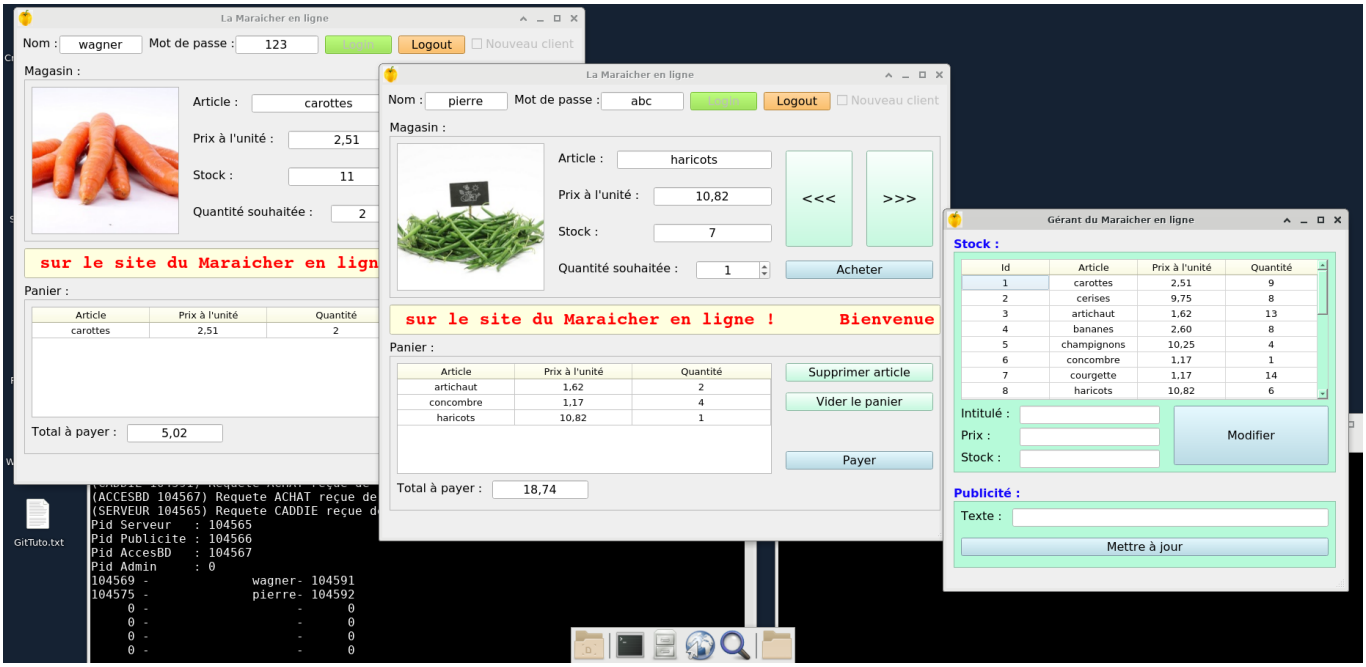
Projet Final UNIX 2022 : « Le Maraicher en ligne »

Consignes :

- Ce projet doit être réalisé par une équipe de 2 étudiants.
- Il sera évalué en janvier, par un professeur responsable, sur la machine virtuelle fournie ou sur la machine de l'école.
- Aucun dossier papier ou électronique ne devra être envoyé à votre professeur de laboratoire avant la date de votre examen.

Description générale :

L'application à réaliser est une application d'achats en ligne de fruits/légumes intitulée « Le Maraicher en ligne » :



Les **clients** de l'application doivent se logger à l'aide d'un couple nom/mot de passe stocké dans un fichier binaire. Une fois loggés, ils peuvent naviguer dans le catalogue du magasin en ligne et faire leurs achats. Leurs achats sont stockés dans un panier apparaissant dans le bas de leur interface graphique. Une publicité apparaît en permanence en rouge dans la zone jaune située au milieu de la fenêtre. Cette publicité défile de la droite vers la gauche en permanence qu'il y ait un client loggé ou pas.

Un **gérant** pourra également se connecter à l'application afin de modifier son stock de fruits et légumes et mettre à jour la publicité.

Evidemment, les clients n'accèdent pas eux-mêmes à la base de données. Toutes les actions réalisées par un utilisateur connecté provoqueront l'envoi d'une requête au **serveur** qui la traitera.

Ce qui est fourni : lien GitHub : https://github.com/hepl-dsoo/LaboUnix2022_Enonce

Comme dans les exercices précédents, on vous fournit tous les fichiers sources nécessaires, fichiers qu'il sera nécessaire de modifier en fonction des étapes à réaliser. Notez qu'il y a un programme que vous ne pouvez pas modifier :

- **CreationBD** : qui crée la table UNIX_FINAL en base de données et y ajoute 21 articles de fruits et légumes.

Notez également l'existence d'un fichier important : **protocole.h**. Celui-ci est inclus et donc connus dans tous le programmes constituant l'application. Ce fichier contient la structure d'un message utilisé pour toute communication par file de message :

```
typedef struct
{
  long  type;
  int   expediteur;
  int   requete;
  int   data1;
  char  data2[20];
  char  data3[20];
  char  data4[100];
  float data5;
} MESSAGE;
```

où

- **type** est le type du message : celui-ci correspondra au pid du destinataire du message, à l'exception du serveur pour lequel le type sera mis à 1.
- **expediteur** est le pid du processus qui a envoyé le message.
- **requete** est un entier pouvant prendre comme valeur une des macros définies dans protocole.h (LOGIN, LOGOUT, …)
- **data1**, **data2**, **data3**, **data4** et **data5** sont des « paquets » de bytes dont la signification varie en fonction de la requête (voir étapes successives).

Remarquez que **protocole.h** vous donne également toutes les utilisations possibles de cette structure, ainsi que le sens d'envoi des messages (Cl = Client, S = Serveur, Ca = Caddie, P = Publicite, BD = AccesBD et Ge = Gerant).

Le serveur gère une **table de connexions** permettant de connaître le pid des fenêtres connectées, le nom des utilisateurs loggés et le pid des processus Caddie associés aux clients (voir étape 3). Ceci est géré par les structures :

```
typedef struct
{
  int   pidFenetre;
  char  nom[20];
  int   pidCaddie;
} CONNEXION;

typedef struct
{
  …
  CONNEXION connexions[6];
} TAB_CONNEXIONS;
```

Six fenêtres au maximum peuvent se connecter simultanément. Chaque fenêtre connectée est représentée sur le serveur par la structure CONNEXION et le serveur stocke 6 structures de ce type dans une seule structure **TAB_CONNEXIONS** qui constitue sa **table de connexions**.

La structure CONNEXION contient :

- **pidFenetre** : le pid de la fenêtre qui s'est connectée sur le serveur (0 sinon)
- **nom** : nom de l'utilisateur qui s'est loggé sur cette fenêtre (chaîne vide si aucun utilisateur ne s'est encore loggé sur cette fenêtre)
- **pidCaddie** : pid du processus chargé de gérer le panier d'achats du client connecté (plus de détails à l'étape 3).

Actuellement, dans le code fourni, cette structure est déjà allouée dynamiquement par le serveur et initialisée correctement. Une fonction **afficheTab()** vous est également fournie pour afficher le contenu de cette table à chaque requête reçue (ou quand vous le voudrez).

N'essayez pas de réaliser tout le projet d'un coup… On vous demande de réaliser ce projet étape par étape.

Etape 0 : Le fichier makefile

On vous demande tout d'abord de créer le **makefile** permettant de réaliser la compilation automatisée de tous les exécutables nécessaires (le fichier **Compile.sh** est fourni). Ceux-ci compilent tous déjà même si aucune fonctionnalité n'a encore été réalisée.

Etape 1 : Connexion/Déconnexion de la fenêtre et login/logout d'un utilisateur

a) Connexion/Déconnexion d'une fenêtre sur le serveur

Lorsque l'application **Client** est lancée, elle doit tout d'abord annoncer sa présence au serveur en lui envoyant son PID. Pour cela, avant même d'apparaître, elle envoie une requête CONNECT au serveur.

A la réception de cette requête, le serveur recherche une ligne vide dans son tableau de connexions et l'insère. Imaginons que la fenêtre client de PID 31195 se lance, le serveur réagit :

```
(SERVEUR 31165) Creation de la file de messages
Pid Serveur 1 : 31165
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0

(SERVEUR 31165) Attente d'une requete...
```

```
(SERVEUR 31165) Requete CONNECT reçue de 31195
Pid Serveur 1 : 31165
 31195 -                    -      0
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0

(SERVEUR 31165) Attente d'une requete...
```

Une fois la fenêtre apparue, un clic sur la croix de la fenêtre

1. envoie une requête DECONNECT au serveur qui supprime la fenêtre de sa table de connexions.
2. termine le processus Client.

b) Login/Logout d'un utilisateur

Un utilisateur peut donc à présent entrer en session en encodant son nom et son mot de passe :

Vous retrouvez ici l'interface graphique de l'exercice 2. La même logique est utilisée ici à part le fait que c'est le serveur qui gère le fichier des clients avec le module « FichierClient » que vous avez développé lors de l'exercice 2.

Donc, un clic sur le bouton « Login »

1. Envoie une requête LOGIN au serveur, celle-ci contient le nom (data2), le mot de passe de l'utilisateur (data3) et un entier (data1 = 1 ou 0) précisant s'il s'agit d'un nouveau client ou pas.
2. Le serveur vérifie dans le fichier binaire « clients.dat » la présence du client et vérifie son mot de passe, ou alors il en crée un nouveau.
3. Si le mot de passe est bon ou qu'un nouveau client a été créé, le serveur ajoute cet utilisateur dans sa table de connexions à la ligne correspondant au PID de la fenêtre.
4. Le serveur envoie la réponse au client, c'est-à-dire un message dont la requête est LOGIN et contenant 1 ou 0 (data1) selon que le login s'est passé correctement ou pas, ainsi qu'un message texte (data4) destiné à l'utilisateur pour lui fournir le résultat de son login (cette phrase sera affichée dans une boîte de dialogue de la fenêtre client). Le serveur envoie au processus client le signal SIGUSR1 pour le prévenir qu'il lui a envoyé un message.

Par exemple, si l'utilisateur « wagner » a réalisé un login à partir de la fenêtre de PID 31195, le serveur réagit :

```
(SERVEUR 31165) Requete CONNECT reçue de 31195
Pid Serveur 1 : 31165
 31195 -                    -      0
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0
```

```
(SERVEUR 31165) Attente d'une requete...
(SERVEUR 31165) Requete LOGIN reçue de 31195 : --0--wagner--123--
Pid Serveur 1 : 31165
 31195 -               wagner-      0
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0

(SERVEUR 31165) Attente d'une requete...
```

Un clic sur le bouton « Logout » doit envoyer une requête LOGOUT au serveur qui supprime donc l'utilisateur de sa table des connexions. Attention que le PID de la fenêtre est maintenu. En effet, la fenêtre **Client** ne se ferme pas lors d'un logout et doit permettre à un nouvel utilisateur de réaliser un login.

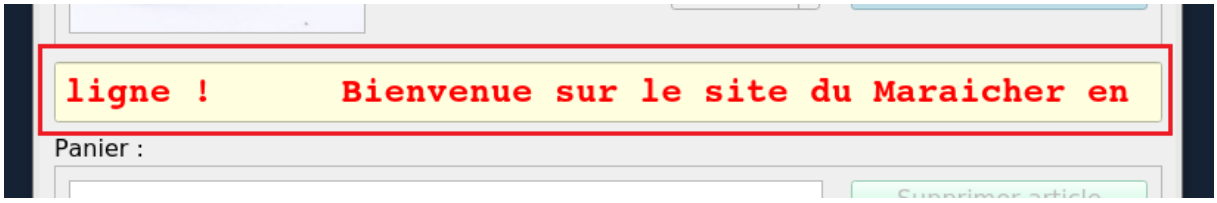
Attention aussi que si un utilisateur clique sur la croix de la fenêtre alors qu'il est loggé, une requête LOGOUT doit être envoyée au serveur avant l'envoi de la requête DECONNECT. Seulement après, le processus **Client** peut se terminer.

c) Suppression des ressources par le serveur

Remarquez que dès à présent, un <CTRL-C> au niveau du serveur doit le faire entrer dans un handler de signal dans lequel il supprime proprement la file de messages.

Etape 2 : Affichage de la publicité dans les fenêtres connectées

Il s'agit à présent de gérer l'affichage d'une publicité au milieu de chaque fenêtre client connectée :



Il n'est pas nécessaire qu'un client soit en session pour que les publicités apparaissent. Dès la connexion de la fenêtre sur le serveur, celle-ci va recevoir les mises à jour des publicités.

Pour l'instant, c'est toujours la même publicité qui est affichée. Cependant, elle défile de manière circulaire dans la zone de texte prévue à cet effet. Cette zone de texte contient 51 cases précisément, dès lors, on se limitera à une publicité dont la chaîne de caractères ne dépasse pas 50 caractères. Toutes les secondes, la publicité est décalée vers la gauche et le caractère (même un espace vide) qui sort par la gauche ré-entre par la droite, il s'agit donc d'un décalage avec rotation.

Les fenêtres clients ne sont pas responsables de la « rotation » de la publicité. Elles doivent récupérer la publicité en l'état à l'aide d'une **mémoire partagée** qui va être une simple **chaîne de caractères (51 caractères + 1 pour le '\0' = 52)**. Pour cela, le serveur

- Doit à présent créer une mémoire partagée dont la clé est celle précisée dans le fichier **protocole.h** (il s'agit de la même clé que la file de messages, ce qui ne pose aucun problème).
- Doit créer le processus **Publicite** dont le rôle va être de réaliser, toute les secondes, la rotation de la publicité dans la mémoire partagée.

Si le texte de la publicité comporte moins de 50 caractères, la mémoire partagée sera complétée de blancs ' ' pour atteindre les 51 caractères (n'oubliez pas le '\0' à la dernière case !!!).

Pour cela, le processus **Publicite** doit

1. Récupérer les identifiants de la file de messages et la mémoire partagée créées par le serveur,
2. S'attacher à la mémoire partagée,
3. Copier le texte de la publicité en mémoire partagée et la « compléter de blancs »
4. Entrer dans une boucle dans laquelle
    a. Il doit prévenir toutes les fenêtres connectées qu'une mise à jour de la publicité doit être affichée (comment ? voir ci-dessous),
    b. Il attend une seconde.
    c. Il réalise le décalage (avec rotation) du contenu de la mémoire partagée (attention que le '\0' doit rester en place…)
    d. Il remonte dans sa boucle.

Pour prévenir toutes les fenêtres connectées, le processus **Publicite** devrait connaître le pid de toutes ces fenêtres. Mais il ne les connaît pas. Par contre, le serveur les connaît ! Donc, après avoir mis à jour la publicité dans la mémoire partagée, le processus **Publicite** doit envoyer une requête UPDATE_PUB au serveur. Dès que le serveur reçoit cette requête, celui-ci parcoure sa table de connexions et envoie le signal **SIGUSR2** à toutes les processus client connectés.

Le processus **Client** doit donc à présent

- Récupérer l'identifiant de la mémoire partagée créée par le serveur,
- Armer le signal **SIGUSR2** sur un handler dans lequel il ira lire le contenu de la mémoire partagée (une chaîne de 51 (+1) caractères donc) et l'afficher dans la fenêtre à l'aide de la **fonction void WindowClient::setPublicite(const char* Text).**

Remarquez que c'est le serveur qui a créé la mémoire partagée. Celle-ci devra être supprimée par programmation par le serveur quand celui-ci se terminera par un <CTRL-C>.

Etape 3 : Affichage des articles et premiers accès à la base de données

Il s'agit à présent de gérer les premiers accès à la base de données contenant les articles du maraicher en ligne, ainsi que leur affichage dans l'interface graphique :

Ce n'est pas le processus **Serveur** qui va accéder directement à la base de données. A chaque client « loggé » va correspondre un processus **Caddie** qui va (dans un premier temps) gérer l'accès à la base de données (mais également la gestion du panier du client correspondant ; voir étape 5). Dans cette étape, le processus Caddie va se contenter de réaliser la recherche en base de données lorsqu'il reçoit une requête de consultation et transmettre le résultat au processus client correspondant.

Dans un premier temps, vous devez compléter le login d'un client. Lorsqu'un client vient de se logger correctement, le processus Serveur crée un processus Caddie (**fork** + **exec**) et stocke le pid du processus Caddie à la ligne correspondante du client dans sa table de connexions :

```
(SERVEUR 58351) Requete CONNECT reçue de 58354
Pid Serveur 1 : 58351
Pid Publicite : 58352
 58354 -                     -      0
     0 -                     -      0
     0 -                     -      0
     0 -                     -      0
     0 -                     -      0
     0 -                     -      0

(SERVEUR 58351) Attente d'une requete...
(SERVEUR 58351) Requete LOGIN reçue de 58354 : --0--wagner--123--
Pid Serveur 1 : 58351
Pid Publicite : 58352
 58354 -              wagner-  58359
     0 -                     -      0
     0 -                     -      0
     0 -                     -      0
     0 -                     -      0
     0 -                     -      0

(SERVEUR 58351) Attente d'une requete...
```

On voit ici que lors du login de wagner, un processus Caddie de pid **58359** a été créé et son pid a été stocké dans la table de connexions à la ligne correspondant à wagner. Ainsi, lors de la réception de futures requêtes provenant de wagner, ces requêtes pourront être transférées au « bon processus » Caddie, c'est-à-dire celui associé à wagner. Le serveur fait office de ce qu'on appelle un « **proxy** ».

Une fois démarré, le processus **Caddie**

- Récupère l'identifiant de la file de message.
- Se connecte à la base de données.
- Entre dans une boucle dans laquelle il attend la réception d'une requête en provenance du serveur (qui l'a reçue du client auquel il est associé).

Lors de la réception d'une requête **LOGIN**, le processus **Caddie** ne fait rien pour l'instant.

Lors de la réception d'une requête **LOGOUT**, le processus **Caddie** se contente pour l'instant de fermer sa connexion à la base de données et se termine par un **exit**.

Il est à présent nécessaire que le serveur gère proprement ses **processus fils (Caddie) zombies**. Vous devez donc armer le signal **SIGCHLD** sur un handler dans lequel le serveur appellera la fonction **wait**. Grâce au pid du fils terminé récupéré, il pourra alors nettoyer sa table de connexions :

```
(SERVEUR 58351) Requete LOGIN reçue de 58354 : --0--wagner--123--
Pid Serveur 1 : 58351
Pid Publicite : 58352
 58354 -              wagner-  58359
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0

(SERVEUR 58351) Attente d'une requete...
(CADDIE 58359) Recuperation de l'id de la file de messages
(CADDIE 58359) Connexion à la BD
(CADDIE 58359) Requete LOGIN reçue de 58354

(SERVEUR 58351) Attente d'une requete...
(CADDIE 58359) Requete LOGOUT reçue de 58354
(SERVEUR 58351) Suppression du fils zombi 58359
(SERVEUR 58351) Attente d'une requete...
(SERVEUR 58351) Requete DECONNECT reçue de 58354
Pid Serveur 1 : 58351
Pid Publicite : 58352
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0
     0 -                    -      0

(SERVEUR 58351) Attente d'une requete...
```

Lorsque le processus client (la fenêtre) reçoit une réponse positive après le log in, il envoie automatiquement une requête **CONSULT** au serveur. Cette requête contient simplement l'**id** (champ data1) de l'article que l'on veut consulter. Lors de cette première requête l'id doit être égal à 1. La requête est transférée au bon processus **Caddie** qui va

- Faire la recherche de l'article en base de données.
- Si la recherche ne donne rien (ce qui n'est pas le cas avec id=1), aucune réponse n'est envoyée. Sinon,
- Envoyer un message contenant toutes les informations (id, intitulé, prix, stock, image) directement au processus client qui a fait la demande.
- Envoyer un signal SIGUSR1 pour prévenir le processus client qu'il a reçu un message.

Dès réception du message, le processus client peut stocker l'article reçu dans sa variable **articleEnCours** (de type **ARTICLE**) et mettre à jour l'affichage dans la fenêtre graphique à l'aide de la **fonction void WindowClient::setArticle(…).**

Il est à présent possible d'implémenter les actions des **boutons « >>> »** (article suivant) et **« <<< »** (article précédent). Pour cela vous devez modifier les fonctions

- **void WindowClient::on_pushButtonSuivant_clicked()**
- **void WindowClient::on_pushButtonPrecedent_clicked()**

Il suffit pour cela d'envoyer une requête **CONSULT** contenant l'**id** de l'article recherché (**articleEnCours.id + 1** pour le suivant et **articleEnCours.id – 1** pour le précédent).

Remarque concernant le format des nombres réels (float) : dans la base de données, les virgules sont représentés par des '.' (Ex : 15.26) tandis que dans l'application C/C++, ces nombres peuvent avoir le symbole ',' pour virgule (Ex : 15,26). Il sera donc parfois nécessaire de réaliser le remplacement d'un symbole par un autre. Un moyen de faire est le suivant :

```
char Prix[20];
sprintf(Prix,"%f",getPrix());
string tmp(Prix);
size_t x = tmp.find(",");
if (x != string::npos) tmp.replace(x,1,".");
```

où ',' a été remplacé par '.'

Etape 4 : Accès unique à la base de données et création d'un pipe

On se rend rapidement compte que la manière de faire décrite plus haut n'est pas optimale :

- Il y a autant de **connexions à la base de données** qu'il y a de processus Caddie en cours d'exécution (c'est-à-dire qu'il y a de clients connectés)
- Lors de futurs achats, il risque d'y avoir des **accès concurrents** à la base de données qui risquent de rendre le stock de marchandises inconsistant.

L'idée est alors de confier l'accès à la base de données à un seul processus **AccesBD**. Tous les processus **Caddie** transmettront alors (via un unique **pipe de communication**) leurs requêtes à **AccesBD** : la connexion à la base de données sera alors unique et il n'y aura plus aucun

accès concurrent. **Le transfert de données via le pipe ne se fait donc que des processus Caddie vers l'unique processus AccesBD**.

Plusieurs choses à mettre en place au niveau du **Serveur** :

- Lors de la création des ressources, le serveur doit créer un **pipe de communication**.
- Lors de la fin du processus serveur, celui-ci doit fermer le pipe en même temps que la suppression des autres ressources.
- Le serveur doit créer le processus **AccesBD** (**fork** + **exec**) et lui passer le descripteur en lecture du pipe lors de l'exec.
- Lors d'un login réussi, le processus Serveur a créé un processus **Caddie** (voir étape 3). Il doit à présent lui passer le descripteur en écriture du pipe lors de l'exec.

Au niveau du processus **Caddie** :

- Il doit récupérer le descripteur en écriture du pipe (**argv**)
- Il ne fait plus aucun accès à la base de données.

Lorsqu'il reçoit (via la file de messages) une requête **LOGIN**, il doit mémoriser le pid du client auquel il est associé.

Lorsqu'il reçoit (via la file de messages) une requête **CONSULT**,

- il modifie le pid de l'expéditeur du message et y place son propre pid.
- Il envoie alors la requête au processus **AccessBD** en l'écrivant sur le pipe de communication. Il **attend alors la réponse** de AccessBD en attendant un message sur la file de messages.
- Dès qu'il reçoit la réponse, il la transmet au processus client correspondant (dont il connaît le pid). Si l'article n'a pas été trouvé, aucune réponse n'est envoyée au processus client.

Au niveau du processus **AccesBD** :

- Il doit récupérer l'id de la file de messages
- Il doit récupérer le descripteur en lecture du pipe (**argv**)
- Il doit se connecter à la base de données.
- Il entre alors dans une boucle dans laquelle il attend de lire une requête sur le pipe.
- Dès qu'il se rendra compte qu'il n'y a plus personne susceptible d'écrire dans le pipe (plus de serveur, ni de processus Caddie), il fermera sa connexion à la base de données et se terminera.

Lorsqu'il reçoit (via le pipe) une requête **CONSULT**,

- Il accède à la base de données (comme le faisait le processus **Caddie** dans l'étape 3)
- Que la recherche fournisse un article ou pas, le processus **AccesBD** va envoyer (via la file de messages) une réponse au processus **Caddie** dont il connaît le pid (champ expediteur de la requête) :
    - Si l'article est trouvé, la réponse contiendra toutes les informations de l'article (id, intitulé, prix, stock, image)
    - Si l'article n'est pas trouvé, le champ data1 (idarticle) de la réponse prend la valeur -1.

Etape 5 : Achats d'articles et mise à jour du panier

Il s'agit ici de gérer l'achat d'articles par le client :

Cette étape va se faire en 2 temps :

a) L'envoi d'une requête d'ACHAT au serveur et la mise à jour de la base de données.
b) L'envoi d'une requête CADDIE au serveur afin de mettre à jour l'affichage du caddie dans la table de la fenêtre du client.

a) Envoi d'une requête ACHAT par le client

Lorsque le client clique sur le bouton « Acheter »

- Une requête **ACHAT** est envoyée au serveur. Cette requête contient l'**id** de l'article en cours (contenu dans la variable **articleEnCours**) et la **quantité** que le client souhaite acheter (récupérée par la méthode **int WindowClient::getQuantite()**).
- Le serveur reçoit et transmet la requête au bon processus **Caddie**.
- Le processus **Caddie** écrit la requête sur le **pipe** à destination du processus **AccesBD** et attend directement une réponse sur la file de messages.
- Le processus **AccesBD** consulte l'article en question dans la base de données :
  - Si le stock est suffisant par rapport à la demande (soit on parvient à acheter tout ce que l'on demande, soit rien du tout) : le stock de la base de données est décrémenté pour l'article en question et la réponse envoyée par **AccesBD** au **Caddie** contiendra (dans le champ **data3** de la réponse) la **quantité achetée** pour cet article.
  - Si le stock est insuffisant par rapport à la demande : aucune mise à jour n'est faite dans la base de données et la réponse envoyée par **AccesBD** au **Caddie** contiendra **« 0 »** (dans le champ **data3** de la réponse) comme quantité achetée pour cet article.
- La réponse est complétée par l'id, l'intitulé, l'image et le prix de l'article avant d'être envoyée au processus **Caddie** correspondant via la file de messages.
- Le processus **Caddie** reçoit la réponse et **se contente actuellement** de la transférer au processus client.

- Le processus **client** affiche alors dans une boîte de dialogue un message du genre « 3 unité(s) de bananes achetées avec succès » si l'achat a été possible ou « Stock insuffisant ! » sinon.

Actuellement, aucun affichage n'est fait sans la table correspondant au panier.

b) Envoi d'une requête CADDIE par le client

L'ensemble des articles achetés par le **client** ne va pas être stocké au niveau du processus client. En effet, si celui-ci plante pour une raison ou pour une autre, il sera alors impossible de remettre à jour correctement la base de données si la session d'achat n'a pas été clôturée correctement. L'ensemble des articles du panier du client va être stocké au sein du processus **Caddie** qui lui est dédié. Dès lors, la première chose à faire est d'ajouter au processus **Caddie** :

- Un **vecteur d'articles** : ARTICLE articles[10]
- Un entier **nbArticles** représentant le nombre d'articles présent dans le panier du client.

Il faut alors mettre à jour le processus **Caddie** : lorsqu'il reçoit la réponse d'**AccesBD**, si celle-ci correspond à un achat réussi (champ data3 != « 0 »), un article est ajouté dans le **vecteur d'articles** et **nbArticles** est incrémenté de 1. Le reste est identique, la réponse est transmise au client.

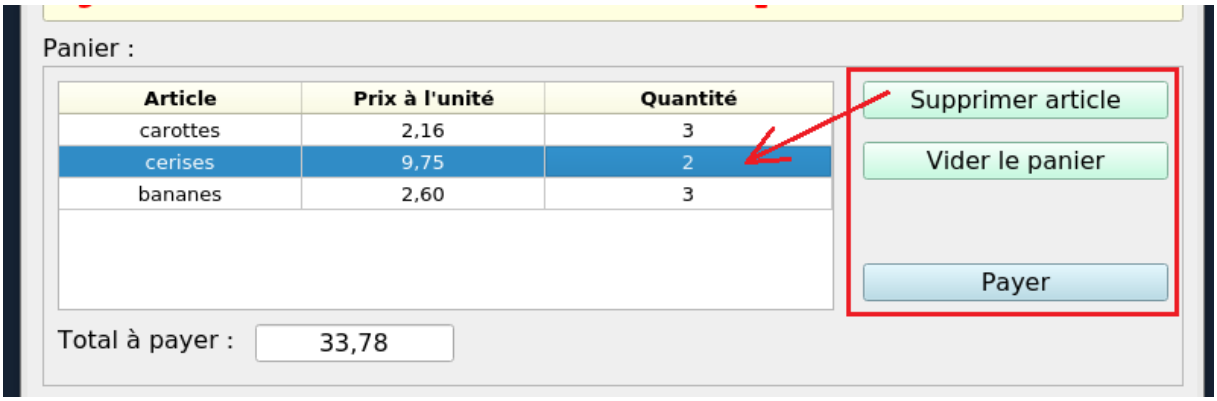
Il faut à présent mettre à jour le processus **client** :

- Lorsqu'il reçoit un message lui disant que l'achat a réussi, en plus d'afficher une boîte de dialogue avec le message de réussite, il envoie une requête **CADDIE** au serveur (cette requête ne contient aucune data autre que le type de la requête). Cette requête a pour but de demander que **l'entièreté du caddie soit retransférer au client**. Le client vide donc la table du panier (méthode **void WindowClient::videTablePanier()**) et met à 0 une variable globale **totalCaddie** ; cette variable correspond au total à payer.
- Le serveur reçoit la requête **CADDIE** et la transmet au bon processus **Caddie**.
- Quand le processus **Caddie** reçoit cette requête, il entre dans une **boucle** dans laquelle il envoie **un message par article** contenus dans son vecteur d'articles. A chaque tour de boucle, il envoie donc un message sur la file de messages et envoie le signal SIGUSR1 pour prévenir le client qu'il a reçu un message.
- A chaque réception d'un tel message, le client ajoute une ligne dans la table du panier (méthode **void WindowClient::ajouteArticleTablePanier(const char *article, float prix, int quantite)**), met à jour sa variable **totalCaddie** et l'affiche dans l'interface graphique (méthode **void WindowClient::setTotal(float total)**)

On remarque bien que le contenu du caddie « n'est pas réellement stocké » dans le processus client (il y est juste affiché).

Etape 6 : Suppression d'articles, paiement et mise ne place d'un Time Out

Il s'agit ici de gérer la suppression d'articles du panier par le client ainsi que le paiement :



Lors d'un clic sur le bouton **« Supprimer article »**, cela a pour effet de supprimer complètement l'article du panier et remettre à jour la base de données :

- On récupère tout d'abord l'**indice de l'article** à supprimer dans la table (utilisation de la méthode **int WindowClient::getIndiceArticleSelectionne()**). Si aucun article n'est sélectionné (retour -1 de la méthode), on affichera une boîte de dialogue d'erreur.
- On envoie une requête **CANCEL** au **serveur**. Cette requête contiendra **l'indice de l'article** à supprimer du panier (champ data1). Le processus client envoie ensuite une requête **CADDIE** au serveur afin de pouvoir faire une mise à jour de l'affichage du panier.
- Comme d'habitude à présent, le serveur transmet la requête au bon processus **Caddie**.
- Connaissant l'indice de l'article, le processus **Caddie** transmet une requête **CANCEL** au processus **AccesBD** (via le pipe). Cette requête contient l'**id de l'article** (champ data1) et la **quantité** correspondante (champ data2). Le processus **Caddie** peut alors supprimer l'article de son vecteur d'**articles** et décrémenter sa variable **nbArticles** de 1.
- Le processus **AccesBD** peut alors faire un accès à la base de données et réincrémenter le stock de l'article correspondant.

Lors d'un clic sur le bouton **« Vider le panier »**, cela a pour effet de vider le panier sans payer (on abandonne donc l'achat) et le stock sera remis à jour en base de données :

- On envoie une requête **CANCEL_ALL** au **serveur**. Le processus client envoie ensuite une requête **CADDIE** au serveur afin de pouvoir faire une mise à jour de l'affichage du panier.
- Le serveur transmet la requête au bon processus **Caddie**.
- Le processus **Caddie** envoie alors au processus **AccesBD** autant de requêtes **CANCEL** qu'il y a d'articles dans son vecteur d'articles. Il peut alors vider son vecteur d'**articles** et remettre sa variable **nbArticles** à 0.

Lors d'un clic sur le bouton **« Payer »**, cela a pour effet de valider définitivement l'achat des articles du panier :

- On envoie une requête **PAYER** au **serveur** et celui-ci se contente à nouveau de la transmettre au bon processus **Caddie**.

- Le processus **Caddie** vide alors le vecteur d'**articles** en remettant **nbArticles** à 0. Pour simplifier l'application, rien n'est fait d'autre concernant le paiement.

IMPORTANT : Remarquez que si l'utilisateur réalise un **LOGOUT** ou quitte l'application cliente alors que le panier contient des articles, la base de données doit être remise à jour !

Mise en place d'un Time Out

Si le client reste en inactivité trop longtemps (nous dirons **60 secondes**), il doit être automatiquement deloggé (l'application cliente doit continuer à tourner et permettre un nouveau login). De plus, si des articles sont présents dans le panier lors du time out, ils doivent être remis en base de données.

Autre événement qui peut se produire : l'application client plante sans pouvoir envoyer de requête au serveur. Il est nécessaire que la base de données soit mise à jour malgré tout.

Pour toutes ces raisons, le **time out** ne peut pas être géré au niveau du processus client. Un time out doit être mis en place dans le processus **Caddie**. Si ce processus ne reçoit pas de requête pendant plus de 60 secondes, il doit recevoir le signal **SIGALRM** (utilisation de **alarm**). Attention que dès que le processus **Caddie** reçoit une nouvelle requête, il doit annuler l'alarme (utilisation de **alarm(0)**).

Lors de la réception du signal **SIGALRM**, le processus **Caddie**

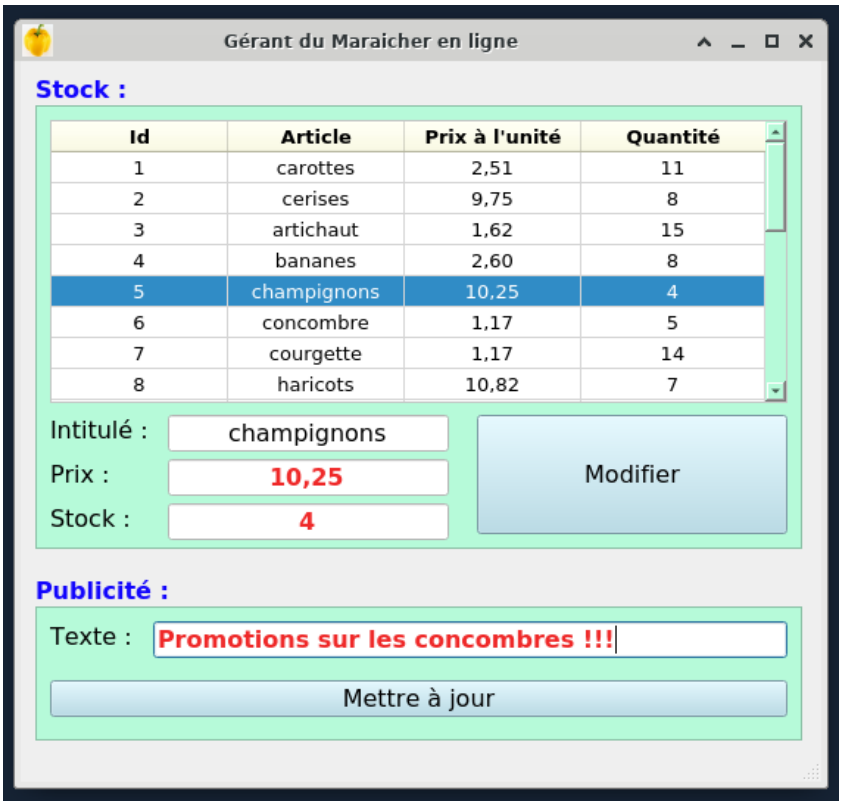
- Remet à jour la base de données en fonction de ce qui se trouve dans le vecteur d'articles (via l'envoi de requêtes **CANCEL** au processus **AccesBD**).
- Envoie une requête **TIME_OUT** au processus **client** auquel il est associé, si celui-ci existe toujours (utilisation de la fonction **kill**).
- Se termine (sa fin réveille le serveur qui pourra mettre à jour sa table de connexions)

Dès réception d'une requête **TIME_OUT**, le processus **client**

- Appelle la méthode **WindowClient::logoutOK()** afin de mettre à jour l'interface graphique.
- Affiche une boîte de dialogue d'erreur disant que le client a été automatiquement déconnecté pour cause d'inactivité.

Etape 7 : Processus Gérant – Mise en place d'un sémaphore

Une application « Gérant du Maraicher en ligne » peut être utilisée pour **modifier les articles** de la base de données (prix et stock uniquement) et mettre à jour la **publicité** qui défile dans les interfaces graphiques des clients. Voici un aperçu de l'interface graphique du processus **Gerant** :



Il s'agit d'une application indépendante qui devra être lancée en ligne de commande.

a) Modification du prix et/ou du stock des articles de la base de données

Aucun login ni mot de passe n'est nécessaire pour utiliser l'application **Gerant**. Il suffit de la lancer en ligne de commande. Cette application se connecte directement à la base de données. Une fois démarrée, elle récupère directement l'ensemble des articles de la base de données et les affiches dans la table « Stock » de l'interface graphique. Pour cela, vous disposez de la méthode **void WindowGerant::ajouteArticleTablePanier(int id,const char* article,float prix,int quantite)**.

Lorsque l'on clique sur un article dans la table, ses caractéristiques (intitulé, prix et stock) s'affichent automatiquement dans les 3 champs de texte situés à gauche du bouton « Modifier ». Seuls les champs « Prix » et « Stock » peuvent être modifiés. De plus, l'id de l'article sélectionné est mémorisé dans une variable globale **idArticleSelectionne**.

Lors d'un clic sur le bouton « **Modifier** » :

- On récupère le prix et le stock modifié par l'utilisateur (utilisation des méthodes **float WindowGerant::getPrix()** et **int WindowGerant::getStock()**)

- On met directement à jour la base de données pour l'article sélectionné.
- On récupère à nouveau l'ensemble des articles dans la base de données afin de mettre à jour l'interface graphique.

b) Empêcher les clients de modifier/consulter la base de données si Gérant est actif

Lorsque le processus **Gérant** est lancé, on doit empêcher les clients de se logger, de consulter ou de modifier le stock en réalisant de achats. Les clients doivent donc être prévenus que le « serveur est en maintenance ».

Pour cela, nous allons utiliser un **sémaphore** valant **1** si la base de données est accessible par les clients et **0** sinon (c'est-à-dire si le gérant est actif). C'est le processus **Serveur** qui doit créer et initialiser correctement le sémaphore. Ce sera également à lui de le supprimer lors de sa fin.

Au niveau du processus **Gérant**,

- Au moment où il démarre, il doit réaliser une **tentative de prise bloquante du sémaphore**. Un fois cette prise réalisée, il garde ce sémaphore, ce qui doit empêcher les clients de réaliser des consultations et/ou des achats d'articles.
- Au moment où le processus **Gérant** se termine (par un clic sur la croix de la fenêtre), le processus Gérant « relâche » le sémaphore, ce qui permettra aux clients de reprendre leurs activités.

Au niveau du processus **Serveur**,

- Lorsqu'il reçoit une requête CONNECT, DECONNECT, UPDATE_PUB ou NEW_PUB (voir ci-après), rien ne doit changer pour lui, la requête doit être traitée normalement.
- Lorsqu'il reçoit toute autre requête, il doit vérifier que le Gérant n'est pas actif. Pour cela, il réalise une **tentative de prise non bloquante du sémaphore** :
  - S'il y arrive (le gérant n'est donc pas actif), la requête est traitée normalement. Ensuite, le sémaphore doit être relâché.
  - S'il n'y arrive pas (le gérant est donc actif), la requête ne peut pas être traitée. En réponse, il envoie une requête **BUSY** au processus client.

Au niveau du processus **client**,

- A la réception d'une requête **BUSY**, il doit afficher une boîte de dialogue d'erreur affichant un message d'erreur du genre « Serveur en maintenance, réessayez plus tard… Merci. »

Afin de simplifier le développement, on ne se tracassera pas des « effets de bords » comme un time out ou un logout survenant alors que le Gérant est actif.

c) Modification de la publicité dans la fenêtre des clients

La publicité peut être mise à jour par le Gérant à tout moment et s'afficher de manière synchronisée dans toutes les fenêtres clients connectées au serveur (pas nécessairement loggée).

Pour cela, lors d'un clic sur le bouton **« Mettre à jour »** :

- On récupère la publicité encodée par l'utilisateur (utilisation de la méthode **const char* WindowGerant::getPublicite()**)
- Le processus **Gérant** envoie une requête **NEW_PUB** au **serveur** via la file de messages. Cette requête contient la nouvelle publicité (champ data4)
- Le serveur reçoit cette requête et la transmet (via la file de messages) au processus **Publicité**. Il lui envoie également le signal SIGUSR1 pour le prévenir qu'il a reçu un message **NEW_PUB**.
- Lors de la réception du signal SIGUSR1, le processus lit le message reçu et met à jour la publicité en mémoire partagée. Celle-ci sera alors automatiquement prise en compte par les fenêtres clients connectées.

Bon travail 😊 !