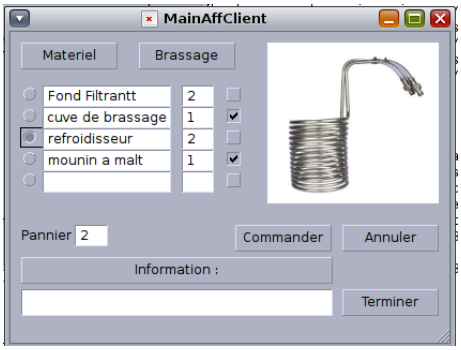
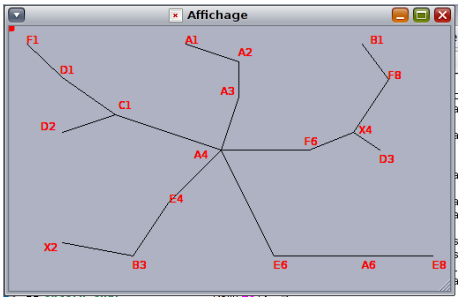
Dossier UNIX 2018.

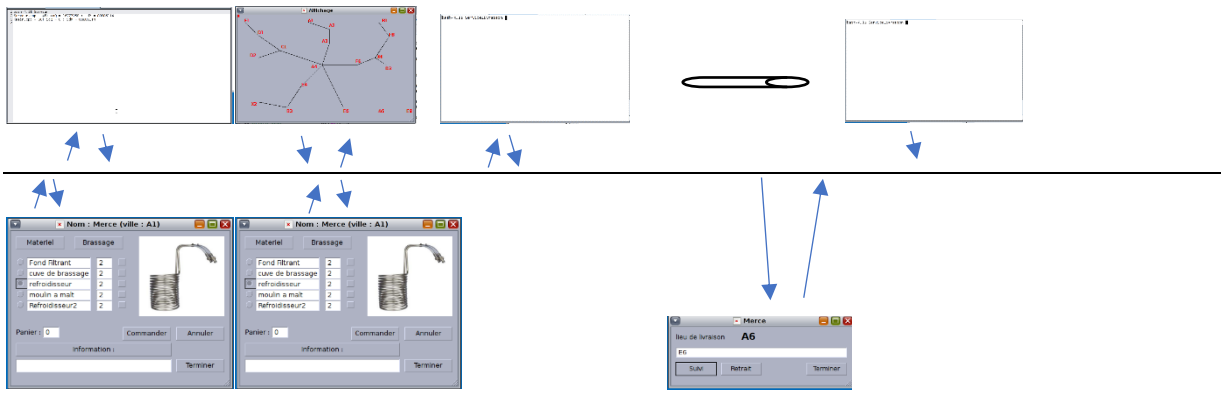
Il s'agit de passer une commande dans un magasin et de se la faire livrer en un lieu choisi.



De plus, une fois la commande passée, on peut suivre son évolution.



Pour cela, on dispose d'une (plusieurs) application(s) **Client** qui communique(nt) avec un processus **Serveur.** Celui-ci gère les différentes commandes dans la totalité.
La communication entre ces processus se fait par file de messages.



En fait, le processus **Client** après s'être identifié au **Serveur**, demande au **Serveur** la liste du matériel disponible ou des produits de brassage.
On en sélectionne certains en fonction du stock, et passe (ou annule) la commande.

Le **Serveur** lui, regarde si le lieu de livraison existe.
Il transmet la sélection faite au **Client.** Il gère le stock en fonction des commandes.
Une fois une commande confirmée, il la transmet au magasin (**ServiceLivraison**) pour effectuer la livraison.

Le processus **Client** peut terminer son exécution si la commande en cours est effectuée ou annulée. **Bref, si aucun achat n'est prévu**.

Il peut évidemment y avoir plusieurs clients en même temps qui effectuent des achats.
De plus, un processus **SuiviCommande** permet à un client de suivre l'évolution de sa commande. (savoir où elle en est dans la livraison, si elle est disponible à l'endroit désigné pour la livraison)

Pour réaliser l'application , on dispose, comme toujours, des différents fichiers constituants l'application.
Le fichier **Compile.sh** :

Que vous connaissez bien.

Le fichier **Commun.ini** :
Qui contient toutes les différentes structures utiles à l'application et les différents #define correspondant aux requêtes.

```
typedef struct { …} MESSAGE;

typedef struct { …} ELEMENT;

typedef struct { …} CLIENT;

typedef struct { …} TABSERVEUR;

#define           CONNECT                       1
#define           CONNECTOK              2
#define           CONNECTINCORRECT       3

#define           MATERIEL               4
#define           BRASSAGE               5
#define           ARTICLECOMMANDE        6
#define           ARTICLEINDISPONIBLE    7
#define           PASSERCOMMANDE         8
#define           ANNULERCOMMANDE        9

#define           PRISEENCHARGE          10
#define           SUIVICOMMANDE          11
#define           IDENTIFICATIONSUIVI    12
#define           COMMANDELIVREE         13
#define           RETRAITCOMMANDE        14
```

Le fichier **Donnee.dat** :

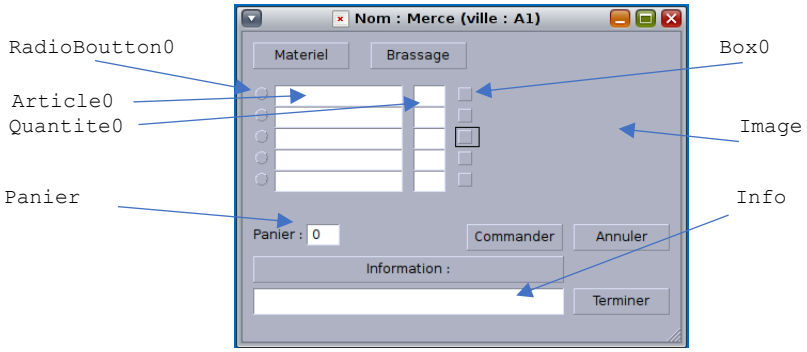
Qui contient les données proprement dites.

```
DONNEE Donnee[] =
   { {"",0,0,0,0},
     {"A1",200,20,200,20},      // 1
      …} ;
…
```

Le processus **Serveur** :
Qui crée toutes les ressources utiles pour l'application, et qui ensuite attend les différentes requêtes.

Le processus **Client** :
Qui peut consulter les articles disponibles, effectuer une commande, …
Toutes les méthodes permettant d'accéder aux variables sont données dans **fctUtilesClient.cpp**.

```
void setArticle0(const char*);…, void setPanier(int Q);
void setQuantite0(int ); int getQuantite0() const; … ,
void setBox0(bool); … , bool getBox0() const; … ,
void setRadioBoutton0(bool);
void setImage(const char* ); void setInfo(const char*);
```



On donne également 2 méthodes :
void clearArticle() qui efface tous les articles.
void setArticle(int) qui affiche les articles en fonctions du nombre d'articles reçus.
Cette méthode affiche l'image du premier article et met le radio0 à true.
Cette méthode devra être compétée plus tard.

Les autres processus
Le processus **SuiviCommande** :
Le Processus **Affichage** :
Le processus **LanceCommande** :
Le processus **VehiculeLivraison** :
seront décrits plus tard, lors de leurs utilisations.

Réalisation :

Pour arriver plus ou moins rapidement à la fin de l'application, bien suivre les étapes.
Chaque étape se base en effet sur le résultat de la précédente.

Etape 1 :

Le **Serveur** crée les ressources nécessaires (c-à-d la file de messages). Cette remarque est valable pour toute l'application.
Le processus **Client** doit se faire connaître au **Serveur**. Cela se fait par la file de messages.

Il transmet une requête **CONNECT** ainsi que son nom et la ville où il veut être livré, ensuite, il se met en attente de la réponse.
Et le **Serveur** répond par une requête **CONNECTOK** ou **CONNECTINCORRECT** selon que la ville existe ou non ou que le nombre de clients est dépassé (pour des raisons pratiques, 4 maximum)
Si la connexion est possible, Le **Serveur** mémorise les renseignements du client.
Il ajoute le nom, la ville et le pid du client dans la structure **TABSERVEUR** .

```
typedef struct
      {
      pid_t             idPid;
      char              NomClient[12];
      char              Ville[12];
      pid_t             idVehicule;
      } CLIENT;

typedef struct
   { pid_t              idServeur1;
     pid_t              idServeur2;
     CLIENT             Cl[MAXCLIENT];
   } TABSERVEUR ;
```

Selon la réponse, le **Client** affiche la fenêtre ou se termine.

Pour la mise au point de l'application, le **Serveur** dispose d'une fonction Affiche() qui permet de vérifier si le TabServeur est bien correct.

IMPORTANT : Lors de l'émission d'un message, ne transmettre que le nombre de bytes nécessaire.

Exemple :

```
bash-4.1$ Client Merce Liege
(mainClient.cpp - 28) Debut
(mainClient.cpp - 54) pas de connexion possible pour le moment
(mainClient.cpp - 55) Ville pas livrée
bash-4.1$

bash-4.1$ Client Merce A2
(mainClient.cpp - 28) Debut
```

Et le fenêtre est lancée.
Exemple d'affichage par le Serveur

```
idServeur1 : 10623
Client :
Cl0 : 10632 -Merce- -A2-         0
Cl1 :         0 -- -- -         0
Cl2 :         0 -- -- -         0
Cl3 :         0 -- -- -         0
```

Une fois cette étape réalisée, vous êtes parti.

Etape 2 :

Le processus **Client** demande au **Serveur** la liste des articles (matériel ou brassage)
Même chose dans le 2 cas, c'est la requête qui diffère (**MATERIEL**, **BRASSAGE**).
Le **Client** effectue la requête.
A la réception de la requête, le **Serveur** lit les articles correspondants.
(5 dans le cas **MATERIEL**, et 4 pour la requête **BRASSAGE**. Il faut bien une limite.
Et il transmet au **Client** un tableau de structure **ELEMENT**.

```
typedef struct
      {
      int         Numero;
      char        Produit[40];
      int         Stock;
      char        Image[40];
      } ELEMENT ;
```

Le **Client** lit la réponse après avoir reçu un signal (**SIGUSR1**).
Il connait le nombre d'éléments en fonction de la taille du message lu.
Il affiche alors la réponse.

Rappel : la fonction **setArticle(n)** ;
Permet d'afficher les n éléments du tableau.

Etape 3 :

Sélectionner un article pour la commande.
Faire la logique dans le premier cas (**Box0**).
Les autres cas seront des copier/coller. (**mais prenez bien le soin de vérifier les résultats**)

Pour cela, il suffit de cliquer sur votre choix(le checkBox0,checkBox1, …)
On possède bien naturellement les méthodes **setBox0(bool)**, **bool getBox0() const**, … .
Lors d'une sélection, le panier est décrémenté ou incrémenté de **1**, selon que l'article avait été sélectionné ou non. Le stock est modifié également.

Il faut mémoriser les articles commandés.
Pour cela, un tableau **int Commande[10]** contient la liste des articles achetés (maximum 10 par client).(On ne peut pas commander plusieurs fois le même article lors d'une commande)

En plus, il faut transmettre au **Serveur** le numéro de l'article ajouté au panier (par +Numero) ou supprimé du panier (-Numero). (**ARTICLECOMMANDE**)
Le **Serveur** saura donc s'il faut incrémenter ou décrémenter le **stock réel** des articles.

Remarque : entre le moment où on affiche les articles et le moment où on effectue la commande, le stock a pu être modifié par un autre **Client**.
Si un client achète un article alors qu'un autre client l'a acheté, le stock réel est alors =0. Le Serveur le signale par une requête **ARTICLEINDISPONIBLE**.
ATTENTION , pas si simple.

Cette étape faite, une modification de la méthode **setArticles(int)** du point précédent est nécessaire. En effet, si le client demande une requête **MATERIEL** (**BRASSAGE**), il faut cocher les articles commandés précédemment. (on peut le faire vu que l'on connait le vecteur **Commande**)

Etape 4 :

Les boutons **Commander** et **Annuler**.

Le bouton **Annuler** :
Remettre le vecteur **Commande**[] à 0 ainsi que le **Panier**.
Effacer tous les articles, vu que le stock est modifié. Pour de nouveau l'afficher, les boutons **Brassage** et **Matériel** sont toujours là.
Transmettre une requête **ANNULERCOMMANDE** au **Serveur** ainsi que le tableau **Commande**[].
Le **Serveur** peut ainsi rectifier le stock des articles (en effet, celui a pu être modifié lors d'une requête précédente).

Le bouton **Commander** :
Remettre le vecteur **Commande**[] à 0 ainsi que le panier.
Effacer tous les articles.
Transmettre une requête **PASSERCOMMANDE** au **Serveur** et le tableau **Commande.**
Le Serveur sait qui a passé la commande par le pid.
Pour le moment, le **Serveur** la reçoit et ne fait rien. (mais pour le moment)

Etape 5 :

Tout cela est très beau, mais un client, c'est peu.
Il faut pouvoir en connecter plusieurs.
Cette étape consiste donc à vérifier que tout est bien correct.
Elle ne doit présenter aucun problème.

Etape 6 :

Le Serveur reçoit la commande et doit la traiter.
Pour cela, il la transmet directement au magasin qui la prépare et l'expédie par l'intermédiaire d'un livreur (éventuellement 2, mais les tests seront plus longs).
C.-à-d. à un processus **ServiceLivraison** qui la transmet par pipe à un processus **VehiculeLivraison**.
La raison est simple, le processus **VehiculeLivraison** ne peut traiter qu'une seule commande à la fois, mais plusieurs commandes peuvent être faite en même temps et doivent être traitées dans l'ordre.
Les processus **ServiceLivraison** et **VehiculeLivraison** sont créés par le **Serveur** en début d'application.

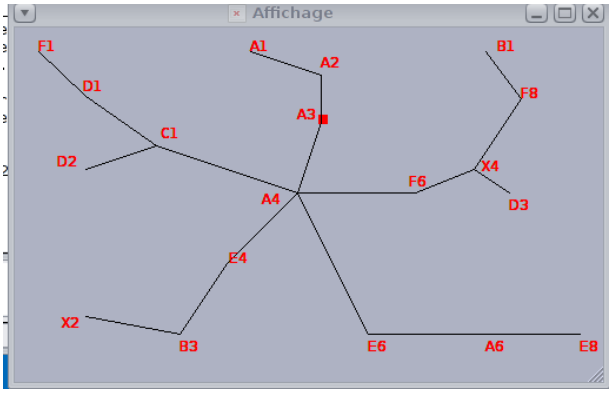
Le processus **ServiceLivraison** ne pose aucun problème. Une fois lancé, il attend (en boucle) un message correspondant à une livraison, (le nom et la ville de livraison) et transmet directement par pipe ces renseignements au processus **VehiculeLivraison.**

Le processus **VehiculeLivraison :**
IL lit dans le pipe.
Dès qu'une commande est lue, il signale au serveur qu'il traite la commande. (commande **PRISEENCHARGE**)
il détermine le tronçon permettant de livrer la ville. (Elle existe forcément, sinon, il n'y a pas de commande.)
et met 10 secondes pour se déplacer d'une étape à l'autre.
un fois la livraison faite, il signale au **Serveur** et il met 5 secondes pour rentrer au magasin (temps ridicule, mais c'est pour les tests)
et il peut lire une éventuelle commande suivante, et ainsi de suite.

Dans le **Serveur**, le client qui effectue une commande a le pid du véhicule = 0.
idServeur1 : 5254
Client:
Cl0 : 5255 -Merce--A2-- 0
Cl1 : 5256 -Wagner--A1-- 0
Cl2 : 0 ----- 0
Cl3 : 0 ----- 0
Dès que le **Serveur** reçoit une requête **PRISEENCHARGE**, il peut alors affecter au client le pid du processus qui effectue la livraison
idServeur1 : 5254
Client:
Cl0 : 5255 -Merce--A2-- 53115
Cl1 : 5256 -Wagner--A1-- 0
Cl2 : 0 ----- 0
Cl3 : 0 ----- 0
Dès que le **Serveur** reçoit une requête **COMMANDELIVREE** (13) il peut alors indiquer au client que sa commande est disponible en mettant le pid du vehicule à 13.
idServeur1 : 5254
Client:
Cl0 : 5255 -Merce--A2-- 13
Cl1 : 5256 -Wagner--A1-- 0
Cl2 : 0 ----- 0
Cl3 : 0 ----- 0
(on peut toujours afficher les résultats pour les vérifier)

Pour faire joli :

Le Serveur lance le processus **Affichage**



Le fichier CompileAffichage.sh est fourni ainsi que les sources.

Il reste à connaître la mémoire partagée pour la communication.

Et le processus **VehiculeLivraison** lui communique sa position.

Cela se fait par mémoire partagée.

A part le fait d'avoir accès à cette mémoire, la logique est fournie en commentaire. Il suffit de compiler.

On peut donc suivre en direct le déplacement du véhicule. (il met 10 (**TEMPS**) secondes pour aller d'une étape à l'autre.) et met 5 secondes pour rentrer en A4.

Etape 7 :

Une fois la commande faite, on peut arrêter le processus **Client**.

mais, bien évidemment, on peut demander où sa commande en est.

C'est le processus **SuiviCommande** qui le permet.

Il communique avec le **Serveur** par une requête **SUIVICOMMANDE.**

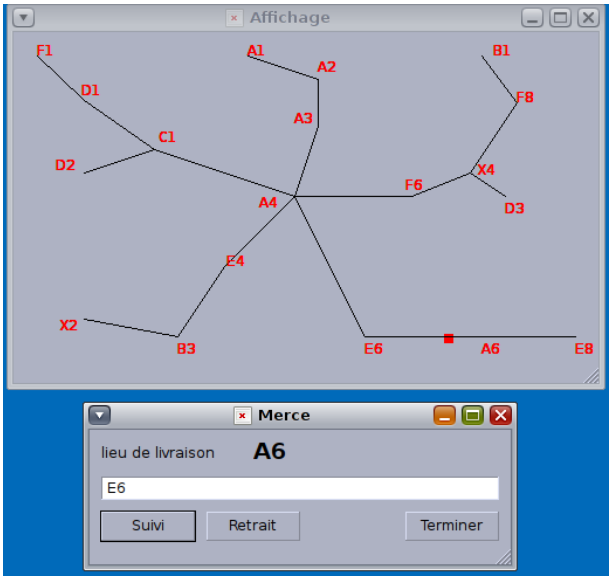
Celui-ci sait, par son TabServeur, si le client a fait une commande (il est présent dans le tableau), si la commande est toujours dans l'atelier (idVehicule = 0)

Si la commande est livrée (idVehicule = 13,

c.à.d **COMMANDELIVREE**) et peut donc répondre directement.

Sinon, il transmet la demande à **VehiculeLivraison** qui répond par la dernière étape réalisée.

Pas de gros problèmes.



Comme d'habitude, la méthode setInfo() est fournie, de même que le fichier CompileSuiviCommande.sh.

Le nom des boutons est évident.

Etape 8 et dernière étape : enfin

On peut lancer 2 **Serveur. MAIS PAS 3**

Et dans ce cas, un des 2 **Serveur** prend en charge la requête, et on ne sait pas lequel.

Si un **Serveur** est à l'arrêt, l'autre continue et l'on ne s'aperçoit de rien.

Un autre **Serveur** peut alors être lancé.

Dans ce cas, si le travail est fait correctement, pas de problème.

Il suffit de placer la structure TabServeur en mémoire partagée, et cela fonctionne parfaitement.

Façon de parler, car il faut également des sémaphores bien placés lorsque l'on modifie les données.

SI VOUS AVEZ PRIS GOUT A L'APPLICATION, ET QUE VOUS AVEZ DU TEMPS

On peut imaginer que 2 utilisateurs différents utilisent l'application.

2 clients différents, 2 administrateurs différents qui lancent un serveur, 1 administrateur et un client différent, …

Cette partie n'est pas obligatoire , c'est juste pour le plaisir.

REMARQUES et RAPPELS.

Ce dossier sera présenté **obligatoirement** sur la machine Sunray2 de la Haute Ecole.

La date de remise du dossier sera le jour prévu pour l'examen.

La pondération est de 50% pour le dossier (les 50% restants seront attribués à la théorie).

Bien évidemment, dans sa réalisation

- La taille des messages envoyés sera la taille nécessaire (et pas la taille maximal)
- L'usage des sémaphores pour les 2 Serveurs est nécessaire.