

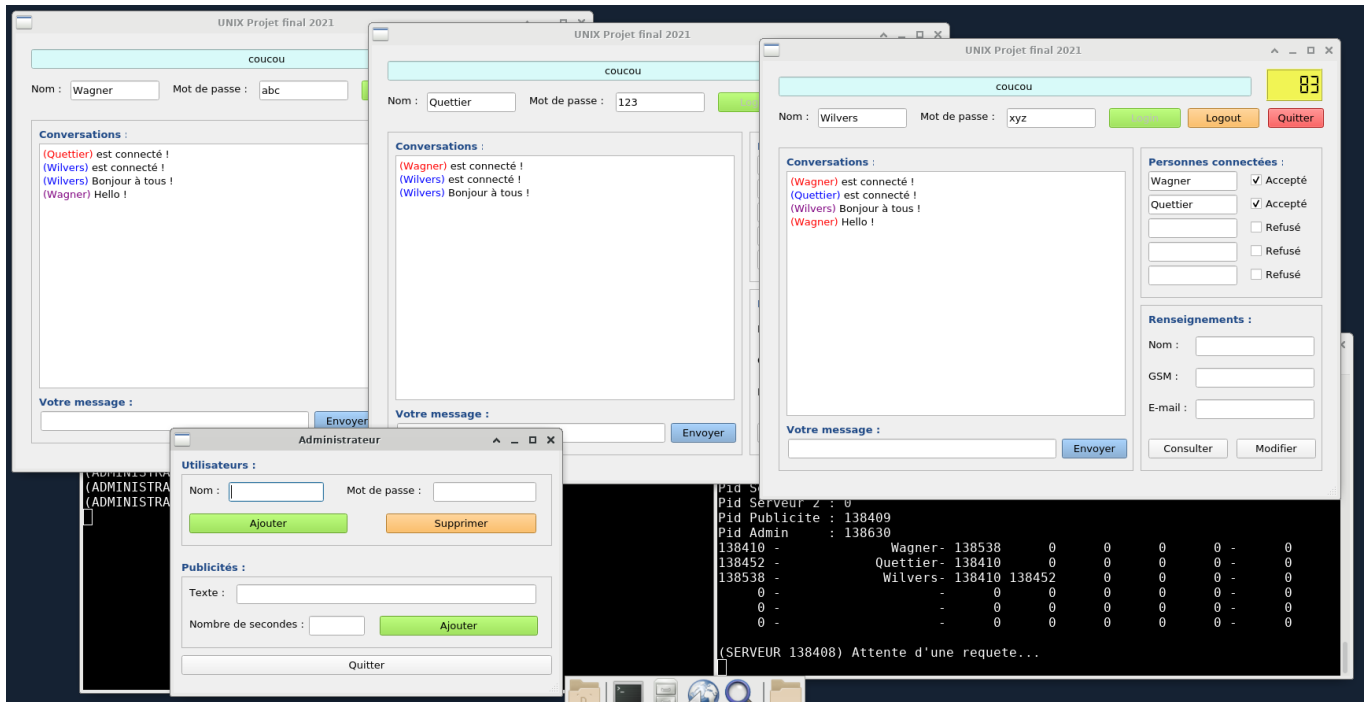
Projet Final UNIX 2021

Consignes :

- Ce projet doit être réalisé par une équipe de 2 étudiants.
- Il sera évalué en janvier, par un professeur responsable, 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 un « chat » multi-utilisateurs :



Les **clients** de l'application doivent se logger à l'aide d'un couple nom/mot de passe stocké en base de données. Une fois loggés, ils voient apparaître les utilisateurs déjà connectés et peuvent choisir ceux avec qui ils souhaitent communiquer. Une publicité apparaît en permanence dans la zone bleue supérieure de la fenêtre. De plus, chaque utilisateur loggé pourra consulter la base de données afin d'obtenir des renseignements (GSM et email) d'autres utilisateurs connus, mais également modifier ses propres données.

Un **administrateur** pourra également se connecter à l'application afin d'ajouter de nouvelles publicités et/ou créer de nouveaux utilisateurs en base de données.

Evidemment, les clients et l'administrateur 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 :

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 2 programmes que vous ne devrez pas modifier :

- **CreationBD** : qui crée la table UNIX_FINAL en base de données et y ajoute 3 utilisateurs de base.
- **BidonFichierPub** : qui crée (bidonne) le fichier **publicites.dat** qui contient 4 publicités de base.

Notez également l'existence d'un fichier important : **protocole.h**. Celui-ci est inclus et donc connu dans tous les 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;
    char   data1[20];
    char   data2[20];
    char   texte[200];
} 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** et **texte** 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 (C = client, S = Serveur, A = Administrateur, Co = Consultation et Mo = Modification).

Le serveur gère une **table de connexions** permettant de connaître le pid des fenêtres connectées, le nom des utilisateurs loggés et le pid des fenêtres avec qui chaque utilisateur souhaite communiquer. Ceci est géré par les structures :

```
typedef struct
{
    int    pidFenetre;
    char   nom[20];
    int    autres[5];
    int    pidModification;
} 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)
- **autres** : pid des fenêtres (maximum 5 forcément) avec qui l'utilisateur loggé accepte de communiquer (plus précisément, d'envoyer ses messages ; voir étape 2)
- **pidModification** : pid du processus chargé de la modification des renseignements de l'utilisateur connecté (plus de détail à l'étape 5).

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 **Compil.sh** est fourni). Ceux-ci compilent tous déjà même si aucune fonctionnalité n'a encore été réalisée.

Etape 1 : Connection/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 77143 se lance, le serveur réagit :

(SERVEUR) Creation de la file de messages

0	-	-	0	0	0	0
0	-	-	0	0	0	0
0	-	-	0	0	0	0
0	-	-	0	0	0	0
0	-	-	0	0	0	0
0	-	-	0	0	0	0

```
(SERVEUR) Attente d'une requete...
(SERVEUR) Requete CONNECT reçue de 77143
77143 -          -          0          0          0          0          0
      0 -          -          0          0          0          0          0
      0 -          -          0          0          0          0          0
      0 -          -          0          0          0          0          0
      0 -          -          0          0          0          0          0
      0 -          -          0          0          0          0          0
      0 -          -          0          0          0          0          0

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

Une fois la fenêtre apparue, un clic sur le bouton « Quitter »

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 :

The image shows a graphical user interface for a login system. It has two input fields: 'Nom' containing 'Wagner' and 'Mot de passe' containing 'abc'. To the right of these fields are three buttons: 'Login' (green), 'Logout' (orange), and 'Quitter' (red). A red rectangular box highlights the 'Login' and 'Logout' buttons.

Pour cela, un clic sur le bouton « Login »

1. envoie une requête LOGIN au serveur, celle-ci contient le nom et le mot de passe de l'utilisateur.
2. Le serveur vérifie en base de données la présence de l'utilisateur et vérifie son mot de passe.
3. Si le mot de passe est bon, 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 « OK » ou « KO » selon que le mot de passe est bon ou pas. Il lui envoie 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 77143, le serveur réagit :

```
(SERVEUR) Requete CONNECT reçue de 77143
77143 -          -          0          0          0          0          0
      0 -          -          0          0          0          0          0
      0 -          -          0          0          0          0          0
      0 -          -          0          0          0          0          0
      0 -          -          0          0          0          0          0
      0 -          -          0          0          0          0          0
      0 -          -          0          0          0          0          0

(SERVEUR) Attente d'une requete...
(SERVEUR) Requete LOGIN reçue de 77143 : --Wagner--abc--
77143 -          Wagner-  0          0          0          0          0
      0 -          -          0          0          0          0          0
      0 -          -          0          0          0          0          0
      0 -          -          0          0          0          0          0
      0 -          -          0          0          0          0          0
      0 -          -          0          0          0          0          0
      0 -          -          0          0          0          0          0

(SERVEUR) 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 le bouton « Quitter » 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 et ferme la connexion à la base de données.

Etape 2 : Liste des utilisateurs déjà connectés, acceptation/refus d'utilisateurs et envoi de messages d'un utilisateur vers les utilisateurs acceptés.

a) Liste des utilisateurs connectés

A partir de maintenant, lorsqu'un utilisateur va se logger, il va recevoir les noms des autres utilisateurs déjà loggés et les afficher dans la partie droite de la fenêtre. De la même manière, lorsqu'un utilisateur va se délogger, les autres clients doivent être prévenus et le faire disparaître de la partie droite de leur fenêtre.

Pour cela, lors d'un LOGIN réussi, le serveur doit envoyer une ou plusieurs requêtes ADD_USER aux utilisateurs déjà connectés mais également à celui qui vient de se connecter. Le champ data1 de la structure du message contiendra le nom de l'utilisateur à ajouter à liste d'un client. Il y a donc un message envoyé du serveur vers le client par client déjà connecté. Imaginons la situation suivante :

```
(SERVEUR) Attente d'une requete...
(SERVEUR) Requete LOGIN reçue de 88084 : --Quettier--123--
88073 -      Wagner-      0      0      0      0      0
88084 -      Quettier-    0      0      0      0      0
88089 -      -          0      0      0      0      0
0 -      -          0      0      0      0      0
0 -      -          0      0      0      0      0
0 -      -          0      0      0      0      0
```

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

Si Wilvers se logge sur la fenêtre 88089, cela donne

```
(SERVEUR) Requete LOGIN reçue de 88089 : --Wilvers--xyz--
88073 -      Wagner-      0      0      0      0      0
88084 -      Quettier-    0      0      0      0      0
88089 -      Wilvers-     0      0      0      0      0
0 -      -          0      0      0      0      0
0 -      -          0      0      0      0      0
0 -      -          0      0      0      0      0
```

(SERVEUR) Attente d'une requete...

et

1. un message ADD_USER contenant Wilvers est envoyé à Wagner (88073)
2. un message ADD_USER contenant Wilvers est envoyé à Quettier (88084)
3. un message ADD_USER contenant Wagner est envoyé à Wilvers (88089)
4. un message ADD_USER contenant Quettier est envoyé à Wilvers (88089)

Lors d'un LOGOUT, le serveur doit envoyer une requête REMOVE_USER à chaque utilisateur restant connecté. Cette requête doit contenir dans son champ data1 le nom de l'utilisateur qui s'est déloggé. Par exemple, si Wagner se délogge,

1. un message REMOVE_USER contenant Wagner est envoyé à Quettier
2. un message REMOVE_USER contenant Wagner est envoyé à Wilvers.

Pour mettre à jour la fenêtre client, vous disposez des méthodes

- **void WindowClient::setPersonneConnectee(int i,const char* Text)**
- **const char* WindowClient::getPersonneConnectee(int i)**

où i est la position d'un utilisateur dans la liste (i=1,2,3,4,5). La méthode getPersonneConnectee retourne une chaîne vide si la case est vide, et pour vider la case, il faut y insérer une chaîne de caractères vide.

b) Accepter/Refuser des utilisateurs

Il s'agit à présent de mettre les utilisateurs loggés en relation en fonction de leurs desiderata. Un utilisateur va en effet pouvoir choisir à qui il souhaite envoyer ses messages. Cela va se faire grâce aux checkbox situés à droite de chaque utilisateur connecté :

Personnes connectées :	
Quettier	<input type="checkbox"/> Refusé
Wilvers	<input type="checkbox"/> Refusé
	<input type="checkbox"/> Refusé
	<input type="checkbox"/> Refusé
	<input type="checkbox"/> Refusé

Par défaut, tous les utilisateurs sont refusés. Cela signifie que l'envoi d'un message (clic sur le bouton « Envoyer » - voir plus loin) provoquera l'envoi d'un message au serveur qui ne le transmettra à personne. Pour changer cela, un clic sur un checkbox dans l'état « Refusé » :

1. mettra le checkbox dans l'état « Accepté » (rien à faire, c'est automatique)
2. enverra au serveur une requête ACCEPT_USER contenant le nom de l'utilisateur à accepter (dans le champ data1).
3. Le serveur met alors à jour sa table de connexions en ajoutant à la bonne ligne (celle de celui qui envoie la requête), dans le champ **autres**, le pid de la fenêtre correspond à l'utilisateur accepté.

Par exemple, imaginons que Wagner, Quettier et Wilvers sont tous les 3 loggés et que :

- Wagner accepte Quettier en envoyant une requête ACCEPT_USER contenant Quettier

- Wagner accepte Wilvers en envoyant une requête ACCEPT_USER contenant Wilvers
- Quettier accepte Wagner en envoyant une requête ACCEPT_USER contenant Wagner
- Wilvers accepte Quettier en envoyant une requête ACCEPT_USER contenant Quettier

L'état de la table des connexions du serveur est alors :

```
(SERVEUR) Attente d'une requete...
(SERVEUR) Requete ACCEPT_USER reçue de 88089
88073 -           Wagner- 88084 88089    0    0    0
88084 -           Quettier- 88073    0    0    0    0
88089 -           Wilvers- 88084    0    0    0    0
0 -           -    0    0    0    0
0 -           -    0    0    0    0
0 -           -    0    0    0    0
```

(SERVEUR) Attente d'une requete...

On remarque que

- Wagner, loggé sur la fenêtre 88073, a accepté d'envoyer des messages aux fenêtres 88084 (Quettier) et 88089 (Wilvers). Tout message provenant de Wagner sera dès lors transmis à Quettier et à Wilvers par le serveur.
- Quettier, loggé sur la fenêtre 88084, a accepté d'envoyer des messages à la fenêtre 88073 (Wagner). Tout message provenant de Quettier sera dès lors uniquement transmis à Wagner par le serveur.
- Wilvers, loggé sur la fenêtre 88089, a accepté d'envoyer des messages à la fenêtre 88084 (Quettier). Tout message provenant de Wilvers sera dès lors uniquement transmis à Quettier par le serveur.

De manière similaire, un clic sur un checkbox dans l'état « Accepté » :

1. mettra le checkbox dans l'état « Refusé » (rien à faire, c'est automatique)
2. enverra au serveur une requête REFUSE_USER contenant le nom de l'utilisateur à refuser (dans le champ data1).
3. Le serveur met alors à jour sa table de connexions en supprimant de la bonne ligne (celle de celui qui envoie la requête), dans le champ **autres**, le pid de la fenêtre correspond à l'utilisateur refusé.

Par exemple, si Wagner refuse Quettier, l'état de la table des connexions du serveur devient alors :

```
(SERVEUR) Attente d'une requete...
(SERVEUR) Requete REFUSE_USER reçue de 88073
88073 -           Wagner- 0 88089    0    0    0
88084 -           Quettier- 88073    0    0    0    0
88089 -           Wilvers- 88084    0    0    0    0
0 -           -    0    0    0    0
0 -           -    0    0    0    0
0 -           -    0    0    0    0
```

(SERVEUR) Attente d'une requete...

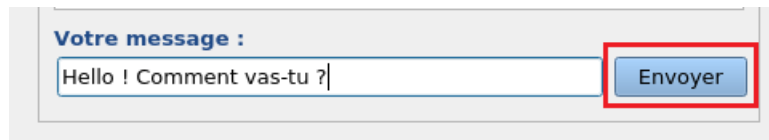
Tout message provenant de Wagner ne sera dès lors plus transmis qu'à Wilvers.

Notez que lors d'un LOGOUT d'un utilisateur :

1. la ligne de la table des connexions du serveur correspondant à cet utilisateur doit être remise à 0,
2. le pid de la fenêtre l'utilisateur déloggé doit également disparaître de la ligne des autres utilisateurs.

c) Envoi de messages d'un utilisateur aux utilisateurs acceptés

Il est maintenant possible d'implémenter le bouton « Envoyer » de la fenêtre client :



Le clic sur ce bouton fait que

1. Le processus **Client** envoie au serveur une requête SEND contenant le texte du message à envoyer (dans le champ texte de la structure message).
2. Le serveur récupère alors le nom de l'utilisateur qui a envoyé le message.
3. Grâce à sa table de connexions, le serveur transfère le message reçu, complété du nom de l'expéditeur dans le champ data1, aux différents utilisateurs autorisés (c'est-à-dire à toutes les fenêtres dont les pid se trouvent dans le champ **autres** correspondant à l'utilisateur émetteur du message).
4. Le serveur envoie le signal SIGUSR1 aux différents destinataires afin de les prévenir qu'un message leur a été envoyé.

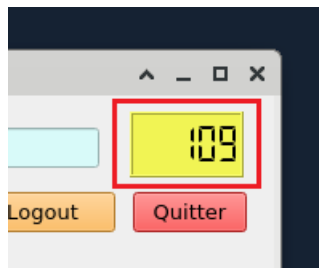
Afin de mettre à jour la partie gauche de la fenêtre affichant les messages reçus des différents utilisateurs, vous disposez de la méthode

void WindowClient::ajouteMessage(const char* personne, const char* message)

qui reçoit en paramètre le nom de l'expéditeur du message et le texte du message. En fonction du nom de l'expéditeur (si celui-ci est présent ou pas dans la liste des utilisateurs connectés), le nom de l'expéditeur est affiché avec une couleur différente selon sa position dans la liste des utilisateurs connectés.

Etape 3 : Gestion d'un Time Out au niveau des utilisateurs connectés

Lorsqu'un utilisateur est loggé et qu'il reste inactif (c'est-à-dire sans cliquer sur le moindre bouton dans la fenêtre) pendant 2 minutes, il doit être automatiquement déloggé. L'utilisateur est informé du temps (en secondes) qui lui reste dans le compteur en haut à droite de la fenêtre :



Pour cela, il faut uniquement modifier le client et non le serveur. Cela se fait grâce à l'utilisation de la fonction **alarm** et du signal **SIGALRM**. Vous devez donc armer le signal **SIGALRM** du processus client sur un handler dans lequel

1. il va décrémenter un compteur global (variable globale de type int appelée `timeOut`)
2. il va mettre à jour l'affichage du compteur dans la fenêtre à l'aide de la fonction **void WindowClient::setTimeOut(int nb)**.
3. il va relancer l'alarme en réalisant un `alarm(1)`.

Le premier `alarm(1)` doit être réalisé dans le constructeur de la fenêtre. C'est également dans le constructeur de la fenêtre que la variable globale `timeOut` doit être initialisée à `TIME_OUT` (macro valant 120).

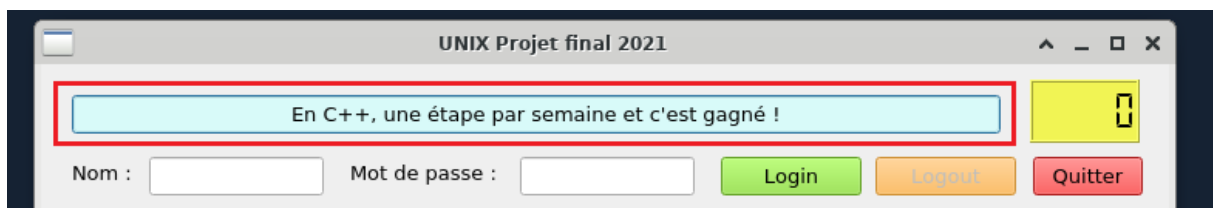
Dans le handler, si le compteur arrive à 0, une requête de **LOGOUT** doit être envoyée au serveur. La fenêtre client doit alors être « resetée », cela peut se faire par appel de la fonction fournie **void WindowClient::logoutOK()**.

A chaque clic sur un bouton/checkbox de la fenêtre, le compteur doit être remis à `TIME_OUT`. Pour cela, à chaque action sur la fenêtre,

1. l'alarme doit être annulée,
2. le compteur `timeOut` doit être remis à `TIME_OUT` et réaffiché dans la fenêtre,
3. une alarme de 1 seconde doit être relancée.

Etape 4 : Affichage des publicités dans les fenêtres connectées

Il s'agit à présent de gérer l'affichage des publicités dans le haut de chaque fenêtre client connectée :



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

Les publicités sont stockées dans un fichier binaire appelé « **publicites.dat** ». Celui-ci peut être bidonné à l'aide de l'exécutable « **BidonFichierPub** » fourni. Ce fichier est un fichier d'enregistrements structurés grâce à la structure **PUBLICITE** définie dans le fichier **protocole.h**. La structure **PUBLICITE** contient

- le texte de la publicité,
- le nombre de secondes que celui-ci doit rester affiché dans les fenêtres connectées jusqu'à l'affichage de la publicité suivante.

Chaque fenêtre va récupérer la publicité en cours à l'aide d'une **mémoire partagée** qui va être une simple **chaîne de caractères (200 caractères)**. 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 lire le fichier **publicites.dat** et placer le texte des publicités en mémoire partagée.

Pour cela, le processus **Publicite** doit

1. récupérer les identifiants de la file de messages et la mémoire partagée créées par serveur,
2. s'attacher à la mémoire partagée,
3. ouvrir le fichier des publicités,
4. entrer dans une boucle dans laquelle
 - a. il lit une publicité dans le fichier. S'il a atteint la fin du fichier, il doit recommencer au début,
 - b. il copie le texte de la publicité dans la mémoire partagée,
 - c. il doit prévenir toutes les fenêtres connectées qu'une nouvelle publicité doit être affichée (comment ? voir ci-dessous),
 - d. il attend le nombre de secondes précisées dans la structure **PUBLICITE** qui vient d'être lue sur disque, avant de remonter 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 copié le texte de 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 parcourt sa table de connexions et envoie le signal **SIGUSR2** à toutes les processus client connectés.

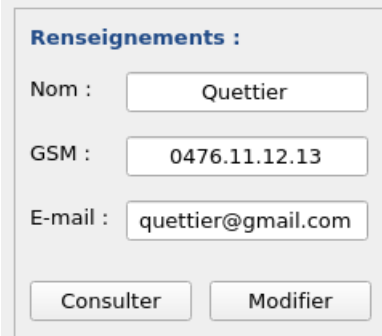
Le processus **Client** doit donc à présent

- 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 200 caractères donc) et l'afficher dans le haut de 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 5 : Consultation/Modification de renseignements par un utilisateur connecté

Nous nous occupons à présent de la partie Modification/Consultation qui ne doit être accessible que par un utilisateur correctement loggé :



Renseignements :

Nom :

GSM :

E-mail :

Un utilisateur peut consulter les informations (GSM et email) de n'importe quel utilisateur (connecté ou non, il suffit qu'il soit présent dans la base de données). Par contre, la modification ne concerne que les informations de l'utilisateur loggé qui pourra alors modifier son mot de passe, son GSM et son email.

a) Consultation de renseignements

Une consultation ne peut pas empêcher le serveur de continuer à répondre aux autres requêtes, dès lors, un processus dédié, appelé **Consultation**, va se charger de répondre à la requête du client.

Un clic sur le bouton « Consulter » va

1. récupérer le **nom** de l'utilisateur dont on désire consulter les renseignements (utilisation de la fonction `const char* WindowClient::getNomRenseignements()`),
2. envoyer au serveur une requête **CONSULT** contenant le nom récupéré, **sans attendre de réponse sur la file de messages (il sera prévenu de l'arrivée des résultats par la réception de SIGUSR1)**.
3. Le serveur va alors créer un processus **Consultation** (le code **Consultation.cpp** vous est fourni en partie) et lui transférer la requête reçue.
4. Le processus **Consultation** va alors pouvoir récupérer les infos en base de données concernant l'utilisateur demandé.
5. Le processus **Consultation** va alors créer la réponse à envoyer au client :
 - Si l'utilisateur est trouvé en BD : « OK » dans le champ data1, le GSM dans le champ data2 et l'email dans le champ texte.
 - Si l'utilisateur n'est pas trouvé en BD : « KO » dans le champ data1.
6. Le processus **Consultation** va envoyer la réponse au processus client dont il connaît le pid et lui envoyer le signal SIGUSR1 pour le prévenir.
7. Le processus **Consultation** se termine.
8. Dans son handler, le processus **Client** récupère les infos et les affiche dans la fenêtre (fonctions `void WindowClient::setGsm(const char* Text)` et `void WindowClient::setEmail(const char* Text)`) ou alors affiche « NON TROUVE » dans le cas où l'utilisateur recherché n'existe pas.

b) Modification de renseignements

Comme pour une consultation, une modification ne peut pas empêcher le serveur de continuer à répondre aux autres requêtes, dès lors, un processus dédié, appelé **Modification**, va se charger de répondre aux requêtes du client.

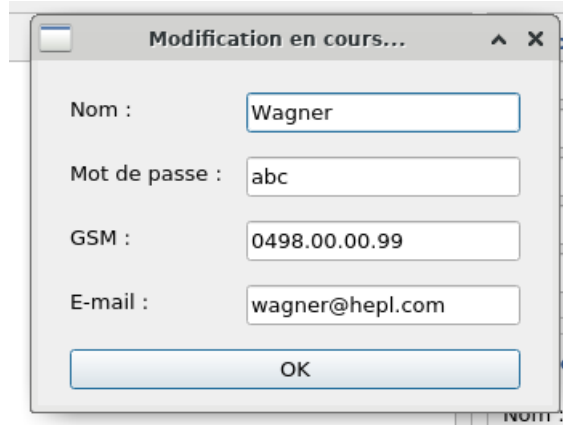
Par contre ici, la modification va se réaliser en 2 étapes qui vont se traduire par l'envoi de 2 requêtes successives au serveur (**MODIF1** et **MODIF2**).

Un clic sur le bouton « Modifier » va

1. Envoyer au serveur une requête **MODIF1**, **et attendre une réponse directement (msgrev)**.
2. Le serveur récupère dans sa table de connexions le **nom** de l'utilisateur qui veut modifier ses renseignements et complète le message reçu du nom de cet utilisateur (dans le champ data1).
3. Le serveur va alors créer un processus **Modification** (le code **Modification.cpp** vous est fourni en partie) et lui transférer la requête reçue. Il stocke alors le pid du processus

créé dans la table des connexions à la ligne correspondant à l'utilisateur qui a fait une demande de modification (champ **pidModification** de la structure CONNEXION).

4. Le processus **Modification** va alors pouvoir récupérer les infos en base de données concernant l'utilisateur demandé (mot de passe, GSM et email).
5. Le processus **Modification** va alors créer la réponse à envoyer au client : mot de passe dans le champ data1, le GSM dans le champ data2 et l'email dans le champ texte.
6. Le processus **Modification** envoie la réponse au processus client dont il connaît le pid et se met en attente de lecture d'une requête **MODIF2**.
7. Le client est débloqué par la lecture de la réponse provenant du processus **Modification**. Il peut alors ouvrir la boîte de dialogue permettant d'afficher les infos reçues et à l'utilisateur de modifier ses données :



8. Lors du clic sur « OK » de la boîte de dialogue, les données modifiées sont récupérées et une requête **MODIF2** contenant le mot de passe modifié, le GSM modifié et l'email modifié est envoyé au serveur.
9. Grâce à sa table de connexions dans laquelle il a stocké le pid du processus **Modification**, le serveur peut transférer à celui-ci la requête reçue. Le serveur sert juste en quelque sorte ici de « proxy ».
10. Le processus **Modification** peut alors modifier la base de données avec les nouveaux renseignements reçus.
11. Le processus **Modification** se termine.

Il est à présent nécessaire que le serveur gère proprement ses **processus fils 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) Attente d'une requete...
(SERVEUR) Requete MODIF1 reçue de 113735
113735 - Wagner- 0 0 0 0 0 - 114757
113740 - - 0 0 0 0 0 - 0
0 - - 0 0 0 0 0 - 0
0 - - 0 0 0 0 0 - 0
0 - - 0 0 0 0 0 - 0
0 - - 0 0 0 0 0 - 0

(SERVEUR) Attente d'une requete...
(MODIFICATION 114757) Recuperation de l'id de la file de messages
(MODIFICATION 114757) Lecture requete MODIF1
(MODIFICATION 114757) Prise non bloquante du sémaphore 0
(MODIFICATION 114757) Connexion à la BD
(MODIFICATION 114757) Consultation en BD pour --Wagner--
```

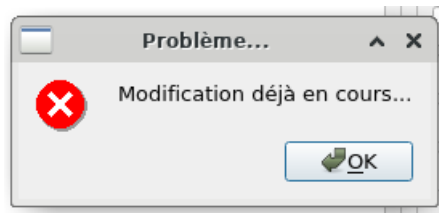
```
(MODIFICATION 114757) Envoi de la reponse
(MODIFICATION 114757) Attente requete MODIF2...
(SERVEUR) Requete MODIF2 reçue de 113735
113735 - Wagner- 0 0 0 0 0 - 114757
113740 - - 0 0 0 0 0 - 0
0 - - 0 0 0 0 0 - 0
0 - - 0 0 0 0 0 - 0
0 - - 0 0 0 0 0 - 0
0 - - 0 0 0 0 0 - 0

(SERVEUR) Attente d'une requete...
(MODIFICATION 114757) Modification en base de données pour --Wagner--
(MODIFICATION 114757) Libération du sémaphore 0
(SERVEUR) Suppression du fils zombi 114757
(SERVEUR) Attente d'une requete...
(SERVEUR) Requete ACCEPT_USER reçue de 113735
113735 - Wagner- 113740 0 0 0 0 - 0
113740 - - 0 0 0 0 0 - 0
```

c) Eviter les accès concurrents à la base de données

Lorsqu'une modification est en cours (la boîte de dialogue de modification étant visible chez un utilisateur),

- Une **consultation est mise en attente** (apparition de « ...en attente... » dans les zones de texte dédiées au GSM et l'email) jusqu'à la fin de la modification en cours. Les résultats pourront ensuite apparaître.
- Une **autre modification doit être refusée** et dans ce cas la boite de dialogue suivante apparaît :



Pour ce faire, un **sémaphore** doit être mis en place. Ce sémaphore devra être créé et initialisé à 1 par le serveur :

- Si le sémaphore est à 1, cela signifie que la base de données est libre d'accès et qu'il n'y a pas de modification en cours.
- Si le sémaphore est à 0, cela signifie qu'une modification est en cours et que la base de données ne peut pas être consultée actuellement (ou modifiée par un autre utilisateur).

Notez dès à présent que c'est le serveur qui devra supprimer proprement le sémaphore en même temps que la suppression de la file de messages et de la mémoire partagée.

Le processus **Consultation** doit être modifié comme suit :

- Il doit faire une **tentative de prise bloquante du sémaphore**. Il doit donc attendre que le sémaphore redevienne positif, et cela juste avant de se connecter à la base de données.
- Il doit relâcher le sémaphore (remise à 1) juste avant de se terminer.

Le processus **Modification** doit quant à lui être modifié comme suit :

- Il doit faire une **tentative de prise non bloquante du sémaphore**, et cela juste avant de se connecter à la base de données. Deux cas sont possibles :

- Le sémaphore est disponible, donc il « le prend » et continue son exécution normalement en accédant à la base de données.
- Le sémaphore n'est pas disponible et n'est pas acquis par le processus. Il doit alors envoyer directement la réponse au client, réponse contenant « KO » dans les 3 champs data1, data2 et texte. Le processus **Modification** se termine alors.
- Il doit relâcher le sémaphore (remise à 1) juste avant de se terminer.

Remarquez qu'il est hors de question que les processus **Client** et **Serveur** manipulent le sémaphore ! La synchronisation ne se fait qu'entre les processus **Consultation** et **Modification** sans altérer le comportement du serveur et des clients connectés/loggés.

Etape 6 : Processus Administrateur

Une application Administrateur peut être utilisée pour ajouter/supprimer des nouveaux utilisateurs dans la base de données mais également pour ajouter des nouvelles publicités au fichier **publicites.dat**. Voici un aperçu de l'interface graphique du processus **Administrateur** :



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

a) Connexion/Déconnexion de l'administrateur sur le serveur

Aucun login ni mot de passe n'est nécessaire pour utiliser l'application Administrateur. Il suffit de la lancer en ligne de commande. Cependant, **une seule application Administrateur ne peut être lancée et utilisée à la fois**. Pour cela, lors du lancement du processus **Administrateur**, celui-ci

1. envoie une requête **LOGIN_ADMIN** au serveur.
2. Le serveur doit posséder une variable **pidAdministrateur** qui contient le pid de l'administrateur déjà connecté. Deux cas peuvent se présenter :
 - Il n'y a pas encore d'administrateur connecté et pidAdministrateur est égal à 0. Le serveur envoie alors un message « OK » (dans le champ data1) au processus **Administrateur** qui peut alors afficher sa fenêtre graphique.

- Un administrateur est déjà connecté et pidAdministrateur est donc différent de 0. Le serveur envoie alors un message « KO » (dans le champ data1) au processus **Administrateur** qui doit se terminer.

Une fois connecté, un clic sur le bouton « Quitter » envoie une requête **LOGOUT_ADMIN** au serveur qui remet simplement sa variable pidAdministrateur à 0. Le processus **Administrateur** se termine alors.

b) Ajout/Suppression d'un utilisateur en base de données

Le processus **Administrateur** **ne modifie pas lui-même la base de données**. Tout doit à nouveau passer par le serveur.

Un clic sur le bouton « Ajouter » dans le cadre Utilisateurs :

1. envoie au serveur une requête **NEW_USER** contenant le nom et le mot de passe de l'utilisateur que l'on veut créer.
2. Le serveur ajoute alors ce nouvel utilisateur en base de données en spécifiant « --- » pour le GSM et l'email. Ceux-ci seront modifiés par l'utilisateur lui-même lorsqu'il rentrera en session.

Un clic sur le bouton « Supprimer » dans le cadre Utilisateurs :

1. envoie au serveur une requête **DELETE_USER** contenant le nom de l'utilisateur que l'on veut supprimer de la base de données. Si celui-ci est actuellement connecté, cela n'a aucun effet sur sa connexion : après son logout, il ne pourra tout simplement plus se logger.
2. Le serveur supprime alors ce nouvel utilisateur de la base de données.

b) Ajout d'une nouvelle publicité dans le fichier publicites.dat

A nouveau, le processus **Administrateur** **ne modifie pas lui-même le fichier publicites.dat**. Tout passe à nouveau par le serveur.

Un clic sur le bouton « Ajouter » dans le cadre Publicités :

1. Envoie au serveur une requête **NEW_PUB** contenant le texte de la publicité et le nombre de secondes qu'elle devra rester affichée dans la fenêtre des utilisateurs.
2. Quand le serveur reçoit la requête, il ouvre le fichier **publicites.dat**. Si celui-ci n'existe pas il est créé.
3. Le serveur se positionne en fin de fichier, écrit la nouvelle publicité et ferme le fichier.

Si le fichier **publicites.dat** n'existe pas au démarrage de l'application, le processus **Publicite**, qui est chargé de le lire et de mettre les publicités lues en mémoire partagée, ne doit pas se terminer pour autant. Il doit attendre que le fichier soit créé par l'administrateur. Pour cela, il se met en attente d'un signal **SIGUSR1** provenant du serveur avant d'essayer de réouvrir le fichier.

Donc, si le serveur doit créer le fichier, il devra envoyer un signal SIGUSR1 au processus **Publicite** pour qu'il se réveille et se mette à lire dans le fichier.

Etape 7 : Doublement du serveur

Afin d'assurer la pérennité de l'application lancée, nous allons gérer ici le fait que **2 serveurs peuvent être lancés simultanément**. Ainsi, si un s'arrête, l'autre continue à recevoir et à traiter les requêtes provenant des clients et de l'administrateur qui doivent ne se rendre compte de rien. Quand un serveur s'arrête, on peut en relancer un second sans aucun souci. On se limitera à 2 serveurs actifs simultanément, ainsi, le lancement d'un 3^{ème} serveur sera refusé.

Notez que les serveurs sont lancés en ligne de commande indépendamment des autres. **Un serveur ne doit pas lancer lui-même un autre serveur !**

Cependant, bien qu'il puisse y avoir 2 serveurs actifs simultanément, il ne peut y avoir qu'un seul processus Publicite et surtout une seule table de connexions communes aux 2 serveurs. Cette table va donc passer en **mémoire partagée** et la mémoire partagée actuelle (simple chaîne de caractères) va être remplacée par une **mémoire partagée structurée** selon la structure

```
typedef struct
{
    char    publicite[200];
    int     pidServeur1;
    int     pidServeur2;
    int     pidPublicite;
    int     pidAdmin;
    CONNEXION connexions[6];
} TAB_CONNEXIONS;
```

Nous y trouvons :

- la publicité mise en mémoire par le processus **Publicite** et lue par les processus clients,
- le vecteur des connexions (qui ne doit donc plus être alloué dynamiquement dans le serveur,
- les pid des serveurs actifs (une valeur à 0 signifie que le serveur n'existe pas),
- les pid des processus **Publicite** (qui reste donc unique) et d'un éventuel **Administrateur** connecté.

Le premier serveur qui est lancé crée les ressources. Le second serveur récupère les identifiants des IPCs déjà créés sans les recréer à nouveau. Plus précisément, lorsqu'on lance un processus serveur :

1. il tente de récupérer l'identifiant de la mémoire partagée :
 - S'il n'y arrive pas, c'est qu'elle n'existe pas et il doit créer toutes les ressources (mémoire partagée, file de messages et sémaphores) et initialiser la table de connexions et les paramètres contenus dans la structure TAB_CONNEXIONS. Il doit aussi créer le processus **Publicite**.
 - S'il y arrive, il doit vérifier que **pidServeur1** ou **pidServeur2** est disponible pour lui (il ne suffit pas de tester que la valeur est égale à 0, il faut aussi tester l'existence du processus car celui a pu être tué de manière brutale). Si les 2 pid sont occupés par des serveur actifs, il doit se terminer. Sinon, il affecte son pid à pidServeur1 ou pidServeur2 selon le cas. Il récupère enfin les identifiants de la file de messages et des sémaphores.

2. il entre dans sa boucle de réception des requêtes.

Remarquez que les deux processus serveurs ne peuvent pas modifier/consulter la table de connexions simultanément. L'accès à celle-ci doit être synchronisée par un **second sémaphore**. Il suffit de créer **un ensemble de 2 sémaphores** :

- Le sémaphore 0 déjà expliqué à l'étape 5c,
- Le sémaphore 1 qui assurera qu'un processus serveur ne pourra pas consulter/modifier la table de connexions pendant que l'autre est actuellement en train de le faire.

N'oubliez pas d'initialiser correctement les 2 sémaphores avant utilisation.

Lors de la fin d'un serveur par un <CTRL-C> (envoi de SIGINT), celui-ci ne doit **supprimer les ressources** (file de messages, mémoire partagée et sémaphores) que s'il est le dernier serveur encore actif. Sinon, il se termine sans supprimer les ressources, après avoir remis à 0 pidServeur1 ou pidServeur2.

Etape 8 (BONUS → non obligatoire donc) : Utilisation de l'application par des users (de la machine jupiter) différents

Dans cette dernière étape, plusieurs utilisateurs (qui ont donc des comptes différents sur la machine jupiter) doivent pouvoir utiliser la même application.

Pour cela, les IPCs devraient être créés de tel sorte qu'ils soient accessibles en lecture/écriture par des utilisateurs du même groupe (student par exemple pour que deux étudiants différents puissent utiliser la même application).

De plus, la **clé** qui est identique pour les 3 types d'IPC et qui est actuellement définie en tant que macro dans le fichier **protocole.h** devrait passer en **paramètre des processus** Serveur, clients et Administrateur. De plus, lors de la création des processus Publicite, Consultation et Modification par un serveur, ceux-ci devraient recevoir, lors de l'**exec**, cette clé en paramètre.

Ainsi, la clé ferait office de port choisi par un groupe d'utilisateurs et plusieurs instances de l'application pourrait co-exister sur la même machine.

Bon travail 😊 !