

NOVEMBRE 2018

PROJET RE216

THÉO LÉPINE & JULIEN MATHET



I. SERVEUR MULTI-CLIENT : ARCHITECTURE **MULTI-THREADÉE**

[SERVEUR] *CRÉATION DES THREADS DÉDIÉS AUX CLIENTS*

[SERVEUR] *TERMINAISON DES THREADS DÉDIÉS AUX CLIENTS*

[CLIENT] *CRÉATION ET TERMINAISON DES THREADS DE COMMUNICATION / RÉCEPTION*

II. GESTION DES UTILISATEURS

III. GESTION DES MESSAGES

IV. GESTION DES SALONS

V. GESTION DES TRANSFERTS DE FICHIERS

I.SERVEUR MULTI-CLIENT : ARCHITECTURE **MULTI-THREADÉE**

- Pourquoi utiliser des threads ?
 - **Flexibilité**
 - **Performances**
 - **Scalabilité**

Utilisation des *pthread* POSIX

Pour le serveur :

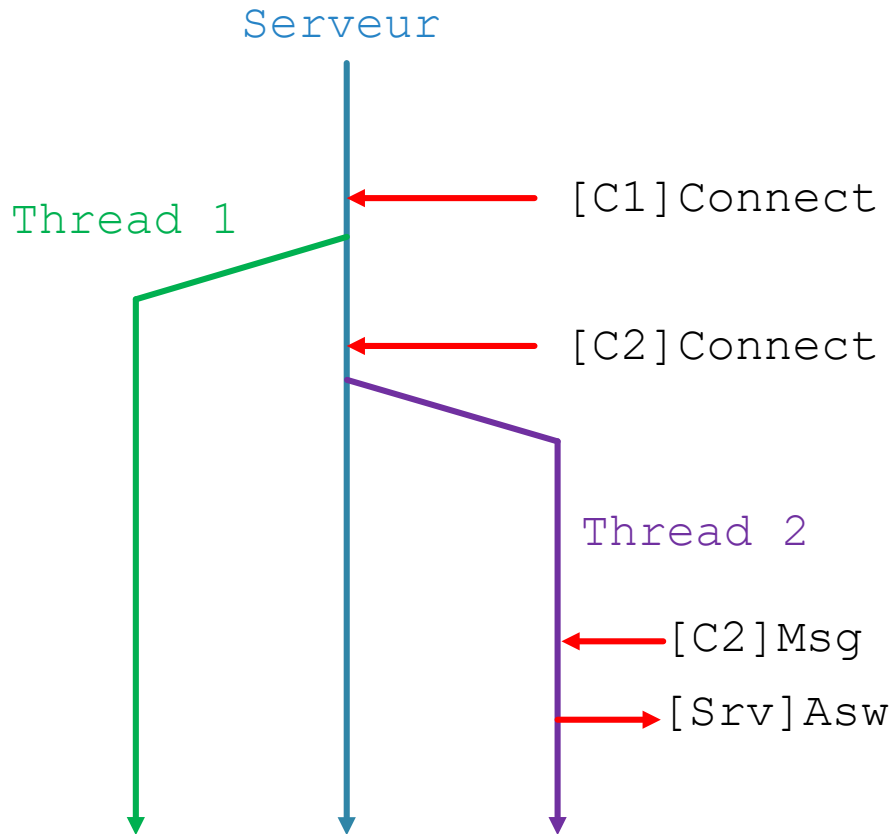
- Un thread par connexion avec un client

Pour le client :

- Un thread de communication (pour l'envoi de messages)
- Un thread de réception (pour la réception et l'affichage)
- Un thread d'envoi un fichier (créé à la demande)

I.SERVEUR MULTI-CLIENT : ARCHITECTURE MULTI-THREADÉE

[SERVEUR] CRÉATION DES THREADS DÉDIÉS AUX CLIENTS



- Lancement du serveur : `./RE216_JALON#_SERVER 8080`

- Mise en place d'une socket d'écoute

- **Accept** des connexions des différents clients

- Vérification si le **nombre d'utilisateurs max** est atteint

- Si check nombre clients OK : **Création d'un thread**

```
pthread_create( &thread, NULL ,  
               connection_handler ,  
               void*)thread_input));
```

Fonction à executer

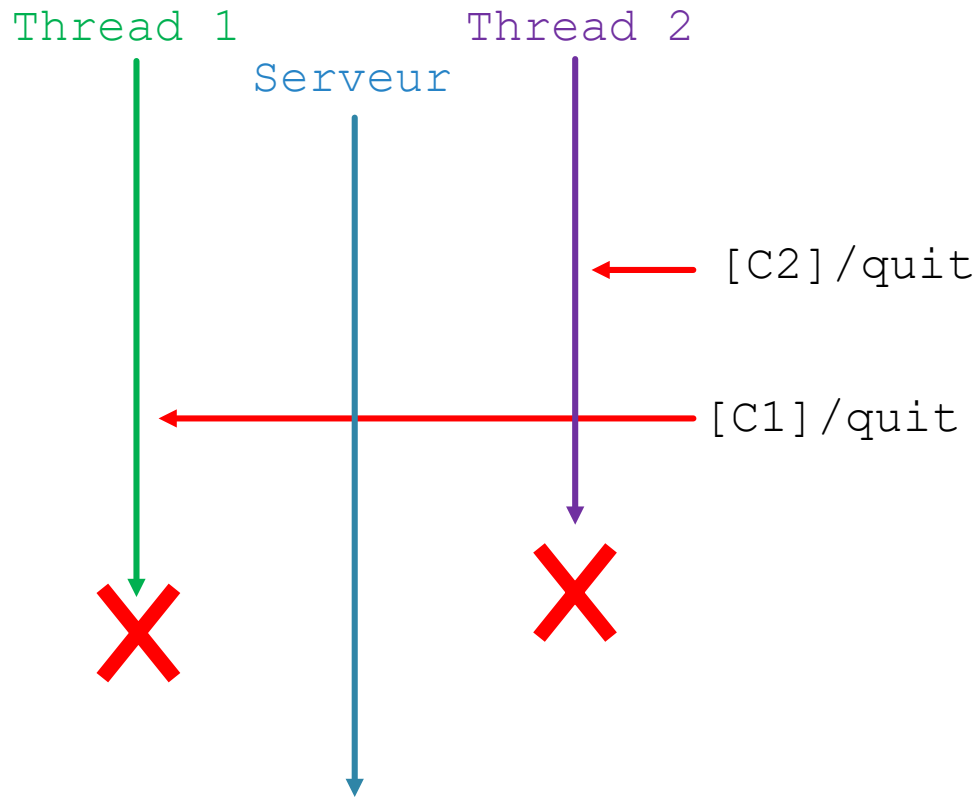
Liste des arguments

- **Détachement** des threads : `pthread_detach(thread);`

→ libération des ressources automatique à la terminaison

I.SERVEUR MULTI-CLIENT : ARCHITECTURE MULTI-THREADÉE

[SERVEUR] TERMINAISON DES THREADS DÉDIÉS AUX CLIENTS



- Les threads clients s'exécutent sous 2 conditions :

```
while (  
    *(thread_args->pt_status) != SERVER_QUITTING  
    && client_status != CLIENT_QUITTING) {...}
```

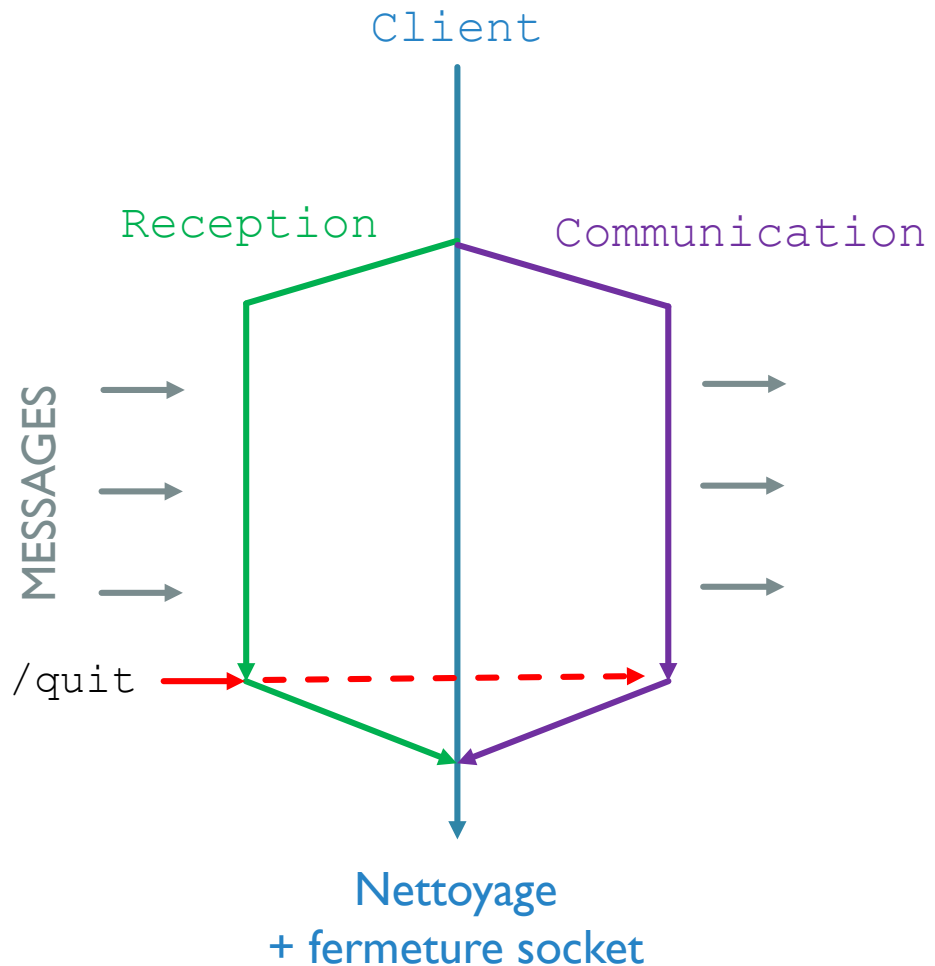
- **/quit** d'un client entraîne :

- Décrémenter le nombre d'utilisateurs
- Changement du status du client `client_status=CLIENT_QUITTING;`
- Suppression de l'utilisateur de la liste chaînée

- Libération des ressources (`free`) et fermeture du `file_descriptor` associé à la socket

I.SERVEUR MULTI-CLIENT : ARCHITECTURE MULTI-THREADÉE

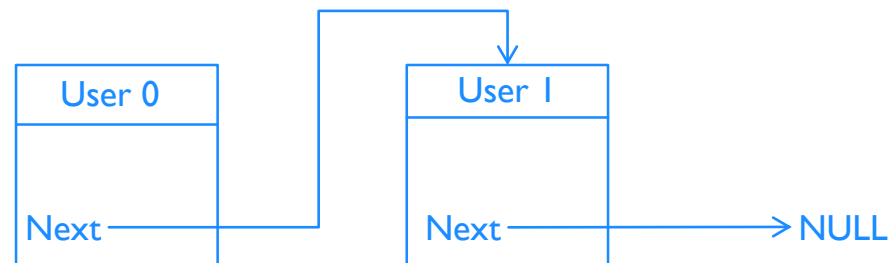
[CLIENT] CRÉATION ET TERMINAISON DES THREADS DE COMMUNICATION / RÉCEPTION



- Lancement d'un client : `./RE216_JALON#_CLIENT 127.0.0.1 8080`
- **Connect** sur la socket sur serveur
- Vérification si le **nombre d'utilisateurs max** est atteint en lisant dans la socket (si `SERVER_FULL` alors `status=CLIENT_QUITTING`)
- Si check nombre utilisateurs OK :
 - **Création d'un thread RECEPTION**
 - **Création d'un thread COMMUNICATION**
- **JOIN** des threads : `pthread_join(reception_thread, NULL);`
 - Attente de la fin des threads clients pour terminer le processus

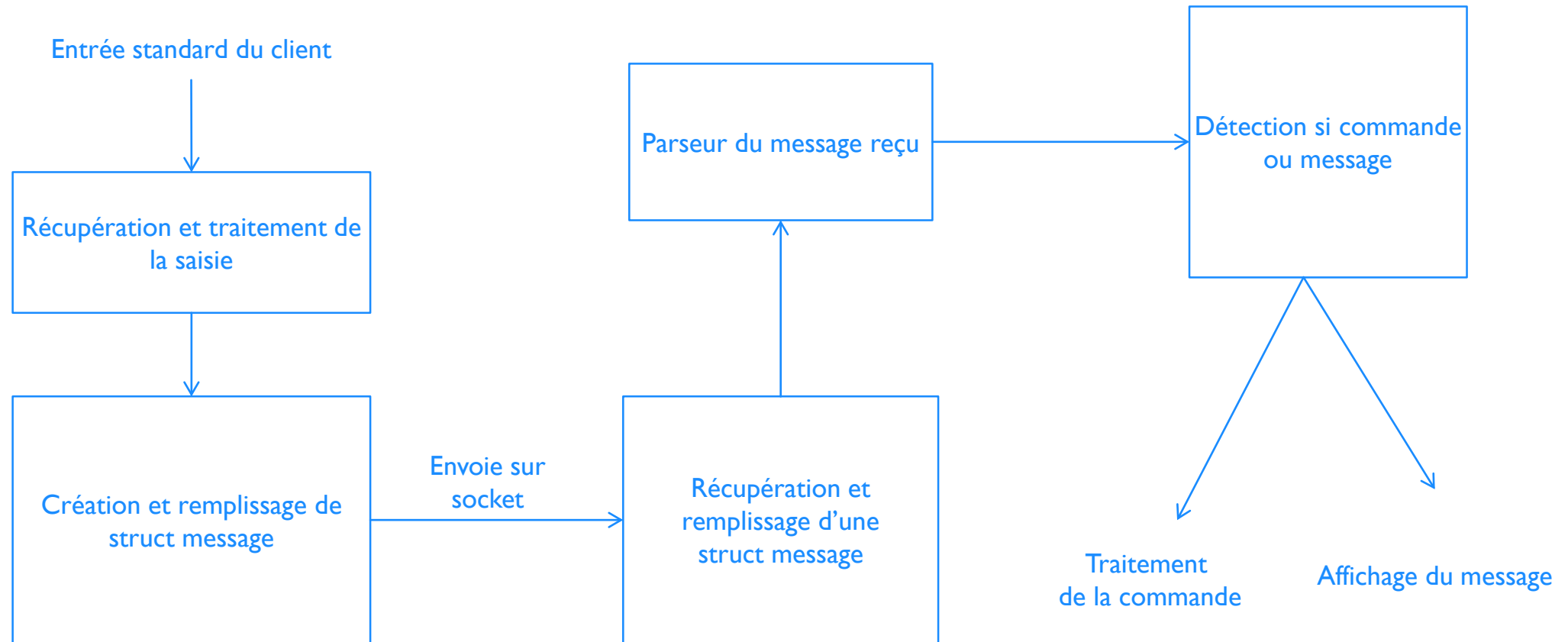
II. GESTION DES UTILISATEURS

- Req 3.1. : `/nick` → **identification obligatoire** avant la création des threads clients → `auth_user(...)`
 - *Check côté client* : `strlen(pseudo) > 1 && *pseudo != '\'` → longueur $\neq 0$ et non vide
 - *Check côté serveur* :
 - **Unicité** du pseudo dans la liste de utilisateurs
 - **Exceptions** : le pseudo ne peut pas être : `Server`, `System` ou `me`
- Req 3.2. : **Gestion des utilisateurs** → infos stockées dans une **liste chaînée** partagée par tous les threads serveurs
- Fonctions `/who` `/whois` fonctionnelles



III. GESTION DES MESSAGES

```
typedef struct message {  
    char * source_pseudo;  
    char * source;  
    char * text;  
} message ;
```



III. GESTION DES MESSAGES

```
typedef struct message {  
    char * source_pseudo;  
    char * source;  
    char * text;  
} message ;
```

La transmission des messages est :

- **Fiable** : pseudo-protocole assurant la transmission totale du message
- **Robuste** : plusieurs vérifications notamment vis-à-vis des appels systèmes
- **Adaptatif** : grâce à un système de buffers, les messages longs peuvent tout de même être transmis

Ceci ce fait via plusieurs fonctions :

- `send_int / receive_int`
- `send_line / receive_line`
- `send_message / receive_message`
- `send_message_to_user`

IV. GESTION DES SALONS (I/2)

```
typedef struct channel {  
    int id;  
    char* name;  
    int members[NB_MAX_CLIENT];  
    int nb_users_inside;  
    struct channel* next;  
}channel;
```

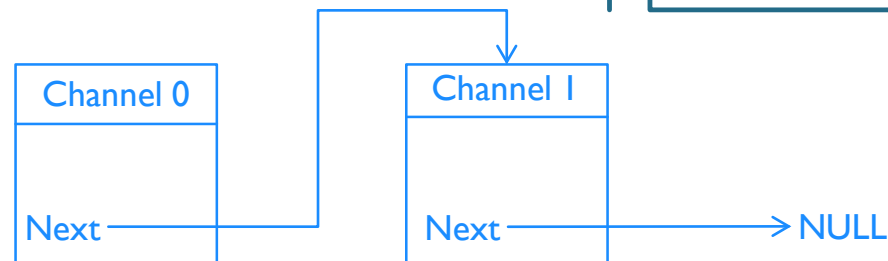
Identifiant unique pour
chaque salon

Nom du salon (unique)

Tableau des membres
(stockage des id des
utilisateurs)

Nombre de membres

→ Suppression auto
lorsque
nb_users_inside=0



IV. GESTION DES SALONS (2/2)

Fonctionnalités salon (jalon 4)

- ✓ **Mutlicast** : Envoi d'un message à tous les utilisateurs du salon
 - ✓ Pas de retransmission à l'expéditeur
 - ✓ Création d'un salon `/create` (unicité du nom)
 - ✓ Rejoindre un salon `/join` (existence du nom et donc du salon)
 - ✓ Quitter un salon (`/quit channel_name` ou `/quit`)
 - ✓ Autodestruction lorsque salon vide
- ✓ **Unicast** : Envoie d'un message privé `/msg`
- ✓ **Broadcast** : Envoi d'un message à tous les utilisateurs `/msgall`

RÉPARTITION DU TRAVAIL LORS DU PROJET

En duo :

- Architecture du projet
- Sockets
- Gestion des utilisateurs
- Débug & Nettoyage global

Théo







Gestion des threads, buffers, messages

Julien

Gestion des salons, du transfert de fichiers

AVANCEMENT DANS LE PROJET

Jalons

- Jalon 1 : modèle client/serveur 
- Jalon 2 : multi-clients 
- Jalon 3 : gestion utilisateurs 
- Jalon 4 : application de chat 
- Jalon 5 : transfert de fichiers 
- Jalon 6 : IPv6 

Améliorations possibles

- Vérification **approfondie** des saisies utilisateurs lors de l'utilisation de commandes (regex, vérifications côté serveur)
- **Sécurité** (authentification, mots de passe, tokens)
- **Fonctions** de chat populaires (MOTD, ops, ban, kill, mute)
- **Interactivité** (utilisation d'API)