Compte Rendu TD 1 : Programmation C et Réseau

Aïman Al-SABAGH-NAHHAS et Alain PIALLAT

18 mars 2025

- `connect` - Connexion à un serveur (TCP)
- `listen` - Écoute des connexions entrantes (TCP)
- `accept` - Accepter une connexion entrante (TCP)
- `write` - Envoi de données (TCP)
- `send` - Envoi de données avec options (TCP)
- `read` - Lecture de données (TCP)
- `recv` - Lecture de données avec options (TCP)
- `shutdown` - Arrêt de la communication sur un socket
- `construire_adresse` - Construction de l'adresse du destinataire

1 - Étude de la fonction tsock

a - Structure de l'appel de la fonction

La commande `tsock` permet de transmettre des messages entre un processus source et un processus puits via les protocoles UDP ou TCP. L'appel de la fonction suit la structure suivante :

```
tsock -p [-options] port
tsock -s [-options] host port
```

Les options disponibles permettent de configurer le comportement du programme :

- `-s` : Définit le programme en mode source (envoi de données vers un puits en attente sur le port donné).
- `-p` : Définit le programme en mode puits (réception de données en attente sur le port donné).
- `-u` : Utilisation du protocole UDP (par défaut, TCP est utilisé si cette option est absente).
- `-l <taille>` : Définit la longueur des données émises ou la longueur maximale des données reçues (par défaut : 30 octets).
- `-w` : Supprime l'affichage des données émises ou reçues.
- `-n <nombre>` : Définit le nombre d'émissions de données pour la source (par défaut : 10) et le nombre de réceptions pour le puits (par défaut : infini).
- `-d` : Impose au service TCP d'envoyer les données sans délai.
- `-t <taille>` : Définit la taille des buffers de transmission du protocole de transport (défaut : taille définie par le système).

b - Description des paramètres

Option	Description	Effet
`-s`	Mode source	Permet d'envoyer des paquets vers une destination
`-p`	Mode puits	Permet de recevoir des paquets sur un port défini
`-u`	Mode UDP	Utilise le protocole UDP au lieu de TCP
`-n <N>`	Nombre de messages	Définit le nombre de messages à envoyer (source) ou à recevoir (puits)

Option	Description	Effet
`-l <L>`	Taille des messages	Définit la longueur des données émises et reçues
`-w`	Suppression d'affichage	Désactive l'affichage des messages émis ou reçus
`-d`	TCP sans délai	Force TCP à envoyer immédiatement les données
`-t <T>`	Taille des buffers	Définit la taille des buffers du protocole de transport

c - fonction à implémenter `construire_adresse`

description

Construit une structure contenant l'adresse IP et le port du destinataire.

Signature en C :

```c
struct sockaddr_in construire_adresse(char* hote, int port);
```

Type	Nom de variable	Explication
`char*`	`hote`	Nom ou adresse IP du destinataire
`int`	`port`	Port cible
`sockaddr_in`	**Retour**	Structure contenant l'adresse et le port formatés
Note		Cette fonction doit être écrite durant le TP car elle ne fait pas partie des bibliothèques standard.

pseudo-code

```
construire_adresse(hote, port):
    adresse = creer une structure sockaddr_in
    adresse.sin_family = AF_INET
    adresse.sin_port = port
    adresse.sin_addr.s_addr = inet_addr(hote)
    retourner adresse
```

---

2 - Implémentation en UDP

a - Description des étapes pour les appels source et puits

Mode source (-s -u)

1. **Créer un socket UDP** avec `socket(AF_INET, SOCK_DGRAM, 0)` :
   - Utiliser `AF_INET` pour spécifier le domaine Internet (IPv4).
   - Utiliser `SOCK_DGRAM` pour indiquer un socket de type datagramme, approprié pour UDP.
   - Le protocole est défini par défaut à UDP en utilisant `0`.
   - La fonction retourne un descripteur de socket (`sock`) si elle réussit, ou -1 en cas d'erreur.
2. **Obtenir les informations de l'hôte** avec `gethostbyname(destination)` :
   - Récupérer les informations de l'hôte à partir du nom de destination.
   - Cette fonction retourne une structure `hostent` contenant l'adresse IP du serveur.
   - **Note** : Cette étape est facultative si l'adresse IP est déjà connue.
3. **Construire l'adresse du destinataire** avec `construire_adresse(host_info->h_addr, port)` :
   - Utiliser les informations de l'hôte pour remplir une structure `sockaddr_in` avec l'adresse IP et le port du serveur.
   - Cette structure est nécessaire pour envoyer des messages au serveur.
4. **Envoyer des messages au serveur** :
   - Pour chaque message à envoyer :
     - Générer un message avec `genere_message()`.
     - Obtenir la taille du message avec `taille(message)`.
     - Utiliser `sendto(sock, message, taille_message, 0, adresse_dest, taille_adresse)` pour envoyer le message.
     - Si l'envoi échoue (retourne -1), attendre avant de réessayer.
5. **Fermer le socket** avec `close(sock)` pour libérer les ressources.

Mode puits (-p -u)

1. **Créer un socket UDP** avec `socket(AF_INET, SOCK_DGRAM, 0)` :
   - Utiliser `AF_INET` pour spécifier le domaine Internet (IPv4).
   - Utiliser `SOCK_DGRAM` pour indiquer un socket de type datagramme, approprié pour UDP.
   - Le protocole est défini par défaut à UDP en utilisant `0`.
   - La fonction retourne un descripteur de socket (`sock`) si elle réussit, ou -1 en cas d'erreur.
   - **Note** : Cette étape est identique à celle du mode source.
2. **Construire l'adresse locale** avec `construire_adresse(INADDR_ANY, port)` :
   - Remplir une structure `sockaddr_in` avec l'adresse IP locale (`INADDR_ANY` pour accepter les connexions sur toutes les interfaces locales) et le port d'écoute.
   - Cette structure est nécessaire pour lier le socket à une adresse locale.
3. **Lier le socket à l'adresse locale** avec `bind(sock, adresse_local, taille_adresse)` :
   - Associer le socket à l'adresse et au port spécifiés.
   - Cette étape est cruciale pour que le serveur puisse recevoir les messages entrants.
4. **Boucle principale pour recevoir des messages** :
   - Lire les messages envoyés par le client avec `recvfrom(sock, message, taille_max, 0, adresse_expediteur, taille_expediteur)`.
   - Afficher ou traiter le message reçu.
   - Continuer à lire jusqu'à ce que le serveur soit arrêté.
5. **Fermer le socket** avec `close(sock)` pour libérer les ressources.

b - Implémentation en pseudo-code

Mode source

```
// Créer un socket TCP
sock <- socket(AF_INET, SOCK_STREAM, 0)

// Obtenir les informations de l'hôte à partir du nom de destination
host_info <- gethostbyname(destination)

// Construire l'adresse du destinataire
adresse_dest <- construire_adresse(host_info->h_addr, port)

// Envoye des messages un par un
répéter N fois :
    genere_message(message) // Générer un message à envoyer
    taille_message <- taille(message) // Obtenir la taille du message
    sendto(sock, message, taille_message, 0, adresse_dest, taille_adresse)
// Envoyer le message

// Fermer le socket après utilisation
close(sock)
```

Mode puits

```
// Créer un socket TCP
sock <- socket(AF_INET, SOCK_STREAM, 0)

// Construire l'adresse locale pour le serveur
adresse_local <- construire_adresse(INADDR_ANY, port)

// Lier le socket à l'adresse locale
bind(sock, adresse_local, taille_adresse)

// Boucle principale pour recevoir des messages
tant que réception active :
    // Lire un message du client
    recvfrom(sock, message, taille_max, 0, adresse_expediteur,
taille_expediteur)
    afficher_message(message) // Afficher ou traiter le message

// Fermer le socket après utilisation
close(sock)
```

3 - Implémentation en TCP

a - Description des étapes pour les appels source et puits

Mode source (-s)

1. **Créer un socket TCP** avec `socket(AF_INET, SOCK_STREAM, 0)` :
    - Utiliser `AF_INET` pour spécifier le domaine Internet (IPv4).
    - Utiliser `SOCK_STREAM` pour indiquer un socket de type flux, approprié pour TCP.
    - Le protocole est défini par défaut à TCP en utilisant `0`.
    - La fonction retourne un descripteur de socket (`sock`) si elle réussit, ou -1 en cas d'erreur.
2. **Obtenir les informations de l'hôte** avec `gethostbyname(destination)` :
    - Récupérer les informations de l'hôte à partir du nom de destination.
    - Cette fonction retourne une structure `hostent` contenant l'adresse IP du serveur.
3. **Construire l'adresse du destinataire** avec `construire_adresse(host_info->h_addr, port)` :
    - Utiliser les informations de l'hôte pour remplir une structure `sockaddr_in` avec l'adresse IP et le port du serveur.
    - Cette structure est nécessaire pour établir la connexion avec le serveur.
4. **Établir la connexion au serveur** :
    - Utiliser `connect(sock, adresse_dest, taille_adresse)` pour se connecter au serveur.
    - Si la connexion échoue (retourne -1), attendre un court moment avant de réessayer.
5. **Envoyer des messages au serveur** :
    - Pour chaque message à envoyer :
        - Générer un message avec `genere_message()`.
        - Obtenir la taille du message avec `taille(message)`.
        - Utiliser `write(sock, message, taille_message)` pour envoyer le message.
        - Si l'envoi échoue (retourne -1), attendre avant de réessayer.
6. **Fermer proprement la connexion** :
    - Utiliser `shutdown(sock, 2)` pour arrêter la réception et l'émission sur le socket.
    - Fermer le socket avec `close(sock)` pour libérer les ressources.

Mode puits (-p)

1. **Créer un socket TCP** avec `socket(AF_INET, SOCK_STREAM, 0)` :
    - Utiliser `AF_INET` pour spécifier le domaine Internet (IPv4).
    - Utiliser `SOCK_STREAM` pour indiquer un socket de type flux, approprié pour TCP.
    - Le protocole est défini par défaut à TCP en utilisant `0`.
    - La fonction retourne un descripteur de socket (`sock`) si elle réussit, ou -1 en cas d'erreur.
2. **Construire l'adresse locale** avec `construire_adresse(INADDR_ANY, port)` :
    - Remplir une structure `sockaddr_in` avec l'adresse IP locale (`INADDR_ANY` pour accepter les connexions sur toutes les interfaces locales) et le port d'écoute.
    - Cette structure est nécessaire pour lier le socket à une adresse locale.
3. **Lier le socket à l'adresse locale** avec `bind(sock, adresse_local, taille_adresse)` :

- Associer le socket à l'adresse et au port spécifiés.
- Cette étape est cruciale pour que le serveur puisse écouter les connexions entrantes sur le bon port.

4. **Mettre le socket en écoute** avec `listen(sock, 5)` :
   - Configurer le socket pour écouter les connexions entrantes.
   - Le paramètre `5` spécifie le nombre maximal de connexions en attente dans la file d'attente.
5. **Accepter les connexions entrantes** :
   - Utiliser `accept(sock, adresse_client, taille_adresse_client)` pour accepter une nouvelle connexion.
   - Cette fonction retourne un nouveau socket (`sock_thread`) dédié à la communication avec le client.
   - L'adresse du client est stockée dans `adresse_client`.
6. **Gérer les connexions client** :
   - Pour chaque connexion acceptée :
     - Lire les messages envoyés par le client avec `read(sock_thread, message, taille_message)`.
     - Afficher ou traiter le message reçu.
     - Continuer à lire jusqu'à ce que le client ferme la connexion.
7. **Fermer le socket client** avec `close(sock_thread)` :
   - Fermer le socket dédié à la communication avec le client une fois la communication terminée.
8. **Fermer le socket principal** avec `close(sock)` :
   - Fermer le socket principal lorsque le serveur n'a plus besoin d'accepter de nouvelles connexions.

b - Implémentation en pseudo-code

Mode source

```
// Créer un socket TCP
sock <- socket(AF_INET, SOCK_STREAM, 0)

// Obtenir les informations de l'hôte à partir du nom de destination
host_info <- gethostbyname(destination)

// Construire l'adresse du destinataire
adresse_dest <- construire_adresse(host_info->h_addr, port)

// Tenter d'établir la connexion jusqu'à ce qu'elle réussisse
tant que connect(sock, adresse_dest, taille_adresse) == -1 :
    attendre // Attendre avant de réessayer

// Boucle principale pour envoyer des messages
tant que non fini :
```

```
    genere_message(message) // Générer un message à envoyer
    taille_message <- taille(message) // Obtenir la taille du message

    // Envoyer le message et attendre l'ACK du serveur
    tant que write(sock, message, taille_message) == -1 :
        attendre // Attendre avant de réessayer

// Fermer proprement la connexion
shutdown(sock, 2)
close(sock)
```

Mode puits

```
// Créer un socket TCP
sock <- socket(AF_INET, SOCK_STREAM, 0)

// Construire l'adresse locale pour le serveur
adresse_local <- construire_adresse(INADDR_ANY, port)

// Lier le socket à l'adresse locale
bind(sock, adresse_local, taille_adresse)

// Mettre le socket en écoute pour accepter les connexions entrantes
listen(sock, 5)

// Boucle principale pour accepter les connexions
tant que non fini :
    // Accepter une nouvelle connexion
    sock_thread, adresse_client <- accept(sock, adresse_client,
taille_adresse_client)

    // Gérer la connexion dans un thread ou un processus séparé
    // (Le pseudo-code ci-dessous montre comment gérer la connexion)

// Fermer le socket principal
close(sock)
```

Gestion des sockets acceptées dans un thread ou un processus séparé

```
// Boucle principale pour lire les messages du client
tant que non shutdown par le client :
    // Lire un message du client
    si read(sock_thread, message, taille_message) == 0 :
        afficher(message) // Afficher ou traiter le message

// Fermer le socket client
close(sock_thread)
```

4 - Annexe

a - structures de données utilisées

`sockaddr_un` - **Adresse de socket dans le domaine Unix**

```
struct sockaddr_un {
    short sun_family; /* = AF_UNIX pour le domaine Unix */
    char sun_path[108]; /* = nom de fichier */
};
```

`sockaddr_in` - **Adresse de socket dans le domaine Internet**

```
struct sockaddr_in {
    short sin_family; /* = AF_INET pour le domaine Internet */
    u_short sin_port; /* = numéro de port */
    struct in_addr sin_addr; /* « comporte » l'@ IP */
    char sin_zero[8]; /* à zéro */
};
```

`sockaddr` - **Structure générale pour les adresses de socket**

```
struct sockaddr {
    u_short sa_family; /* domaine */
    char sa_data[14]; /* référence */
};
```

`in_addr` - **Structure pour l'adresse IP**

```
struct in_addr {
    union {
        struct {...};
        struct {...};
        u_long S_addr; /* = @ IP */
    } S_un;
    #define s_addr S_un.S_addr
};
```

`hostent` - **Structure pour les informations d'hôte**

```
struct hostent {
    char* h_name; /* nom de la station */
    char** h_alias; /* nom des alias */
    int h_addrtype; /* classe A, B ou C */
    int h_length; /* longueur du champ h_addr_list */
    char** h_addr_list; /* liste d'adresse IP */
    #define h_addr h_addr_list[0]; /* 1ère @ de la liste */
};
```

b - Listing des fonctions utilisées

`socket` - Création d'un socket

Crée un socket et retourne son identifiant local.

Signature en C :

```
int socket(int domaine, int type, int protocole);
```

Type	Nom de variable	Explication
`int`	`domaine`	Domaine du socket (`AF_INET` pour IPv4)
`int`	`type`	Type du socket (`SOCK_STREAM` pour TCP ou `SOCK_DGRAM` pour UDP)
`int`	`protocole`	Protocole à utiliser (0 pour le protocole par défaut, `IPPROTO_UDP` pour UDP, `IPPROTO_TCP` pour TCP)
`int`	**Retour**	-1 en cas d'erreur, représentation interne du socket si succès

`close` - Fermeture d'un socket

Ferme un socket.

Signature en C :

```
int close(int sock);
```

Type	Nom de variable	Explication
`int`	`sock`	Représentation interne du socket
`int`	**Retour**	-1 en cas d'erreur

`gethostbyname` - Résolution de nom d'hôte

Convertit un nom de domaine en adresse IP.

Signature en C :

```
struct hostent* gethostbyname(char* nom_station);
```

Type	Nom de variable	Explication
`char*`	`nom_station`	Nom de l'hôte à résoudre
`hostent*`	**Retour**	NULL en cas d'erreur, pointeur vers une structure de type `hostent` contenant l'adresse IP de l'hôte

`getsockname` - Récupération de l'adresse locale d'un socket

Récupère l'adresse locale associée à un socket.

Signature en C :

```
int getsockname(int sock, struct sockaddr* padr, int* plg_adr);
```

Type	Nom de variable	Explication
`int`	`sock`	Représentation interne du socket
`sockaddr*`	`padr`	Pointeur vers une structure de type `sockaddr` qui contiendra l'adresse du socket
`int*`	`plg_adr`	Pointeur vers un entier qui contiendra la longueur de la structure `sockaddr`
`int`	**Retour**	-1 en cas d'erreur, 0 si succès

`bind` - Association d'une adresse à un socket

Associe une adresse et un port locaux à un socket.

Signature en C :

```
int bind(int sock, struct sockaddr* padr, int lg_adr);
```

Type	Nom de variable	Explication

Type	Nom de variable	Explication
`int`	`sock`	Représentation interne du socket
`sockaddr*`	`padr`	Pointeur vers une structure de type `sockaddr` qui contient l'adresse du socket
`int`	`lg_adr`	Longueur de la structure `sockaddr`
`int`	**Retour**	-1 en cas d'erreur, 0 si succès

`sendto` - Envoi de données (UDP)

Envoie un message à une adresse spécifique.

Signature en C :

```
int sendto(int sock, char* pmesg, int lg_mesg, int option, struct sockaddr*
padr_dest, int lg_adr_dest);
```

Type	Nom de variable	Explication
`int`	`sock`	Représentation interne du socket émetteur
`char*`	`pmesg`	Pointeur vers le message à envoyer
`int`	`lg_mesg`	Longueur du message à envoyer en octets
`int`	`option`	Options supplémentaires (0)
`sockaddr*`	`padr_dest`	Pointeur vers une structure de type `sockaddr` qui contient l'adresse du destinataire
`int`	`lg_adr_dest`	Longueur de la structure `sockaddr`
`int`	**Retour**	-1 en cas d'erreur, nombre d'octets envoyés si succès

`recvfrom` - Réception de données (UDP)

Lit les données reçues dans le buffer et récupère l'adresse de l'expéditeur.

Signature en C :

```
int recvfrom(int sock, char* pmesg, int lg_max, int option, struct
sockaddr* padr_em, int* plg_adr_em);
```

Type	Nom de variable	Explication
`int`	`sock`	Représentation interne du socket récepteur
`char*`	`pmesg`	Adresse mémoire à laquelle le message lu sera stocké
`int`	`lg_max`	Longueur maximale du message
`int`	`option`	Options supplémentaires (0)
`sockaddr*`	`padr_em`	Pointeur vers une structure de type `sockaddr` qui contiendra l'adresse de l'émetteur
`int*`	`plg_adr_em`	Pointeur vers un entier qui contiendra la longueur de la structure `sockaddr`
`int`	**Retour**	-1 en cas d'erreur, nombre d'octets reçus si succès

`connect` - Connexion à un serveur (TCP)

Établit une connexion avec un serveur.

Signature en C :

```c
int connect(int sock, struct sockaddr* padr_serv, int lg_adr_serv);
```

Type	Nom de variable	Explication
`int`	`sock`	Représentation interne du socket
`sockaddr*`	`padr_serv`	Pointeur vers une structure de type `sockaddr` qui contient l'adresse du serveur
`int`	`lg_adr_serv`	Longueur de la structure `sockaddr`
`int`	**Retour**	-1 en cas d'échec de connexion, 0 si succès

`listen` - Écoute des connexions entrantes (TCP)

Met un socket en mode écoute pour accepter les connexions entrantes.

Signature en C :

```c
int listen(int sock, int nb_max);
```

Type	Nom de variable	Explication

Type	Nom de variable	Explication
int	sock	Représentation interne du socket
int	nb_max	Nombre maximal de connexions en attente
int	**Retour**	-1 en cas d'erreur, 0 si succès

---

accept - Accepter une connexion entrante (TCP)

Accepte une connexion entrante et crée un nouveau socket pour la gérer.

Signature en C :

```c
int accept(int sock, struct sockaddr* padr_client, int* plg_adr_client);
```

Type	Nom de variable	Explication
int	sock	Représentation interne du socket
sockaddr*	padr_client	Pointeur vers une structure de type sockaddr qui contiendra l'adresse du client
int*	plg_adr_client	Pointeur vers un entier qui contiendra la longueur de la structure sockaddr
int	**Retour**	-1 en cas d'erreur, représentation interne du socket local dédié à la connexion établie si succès

---

write - Envoi de données (TCP)

Envoie un message via un socket connecté.

Signature en C :

```c
int write(int sock, char* pmesg, int lg_mesg);
```

Type	Nom de variable	Explication
int	sock	Représentation interne du socket
char*	pmesg	Pointeur vers le message à envoyer
int	lg_mesg	Longueur du message à envoyer en octets
int	**Retour**	-1 en cas d'erreur, nombre d'octets envoyés si succès

---

send - Envoi de données avec options (TCP)

Envoie un message avec des options spécifiques.

Signature en C :

```c
int send(int sock, char* pmesg, int lg_mesg, int option);
```

Type	Nom de variable	Explication
int	sock	Représentation interne du socket
char*	pmesg	Pointeur vers le message à envoyer
int	lg_mesg	Longueur du message à envoyer en octets
int	option	Options supplémentaires (0 ou MSG_OOB pour données urgentes)
int	**Retour**	-1 en cas d'erreur, nombre d'octets envoyés si succès

read - Lecture de données (TCP)

Lit les données reçues dans le buffer.

Signature en C :

```c
int read(int sock, char* pmesg, int lg_max);
```

Type	Nom de variable	Explication
int	sock	Représentation interne du socket
char*	pmesg	Adresse mémoire à laquelle le message lu sera stocké
int	lg_max	Longueur maximale du message
int	**Retour**	-1 en cas d'erreur, nombre d'octets lus si succès

recv - Lecture de données avec options (TCP)

Lit les données reçues avec des options spécifiques.

Signature en C :

```c
int recv(int sock, char* pmesg, int lg_max, int option);
```

Type	Nom de variable	Explication

Type	Nom de variable	Explication
`int`	`sock`	Représentation interne du socket
`char*`	`pmesg`	Adresse mémoire à laquelle le message lu sera stocké
`int`	`lg_max`	Longueur maximale du message
`int`	`option`	Options supplémentaires (0, `MSG_OOB` pour données urgentes, ou `MSG_PEEK` pour lecture sans effacement)
`int`	**Retour**	-1 en cas d'erreur, nombre d'octets lus si succès

`shutdown` - Arrêt de la communication sur un socket

Arrête la réception et/ou l'émission sur un socket.

Signature en C :

```c
int shutdown(int sock, int sens);
```

Type	Nom de variable	Explication
`int`	`sock`	Représentation interne du socket
`int`	`sens`	Sens de l'arrêt (0 pour arrêter la réception, 1 pour arrêter l'émission, 2 pour les deux)
`int`	**Retour**	-1 en cas d'erreur, 0 si succès

`construire_adresse` - Construction de l'adresse du destinataire

Construit une structure contenant l'adresse IP et le port du destinataire.

Signature en C :

```c
struct sockaddr_in construire_adresse(char* hote, int port);
```

Type	Nom de variable	Explication
`char*`	`hote`	Nom ou adresse IP du destinataire
`int`	`port`	Port cible
`sockaddr_in`	**Retour**	Structure contenant l'adresse et le port formatés

Type	Nom de variable	Explication
Note		Cette fonction doit être écrite durant le TP car elle ne fait pas partie des bibliothèques standard.