Le réseau de l'UFR d'informatique

- On peut faire tourner un client et un serveur sur la même machine $\rightarrow$ on dit que le client et le serveur s'exécutent **localement**.

  ```
  lulu:~$ nc -l 7879          lulu:~$ nc localhost 7879
  ```

  Le serveur et le client s'exécutent sur lulu.

- On peut faire tourner un client sur une machine et un serveur sur une autre machine $\rightarrow$ on dit que le le client et le serveur s'exécutent sur des machines **distantes**.

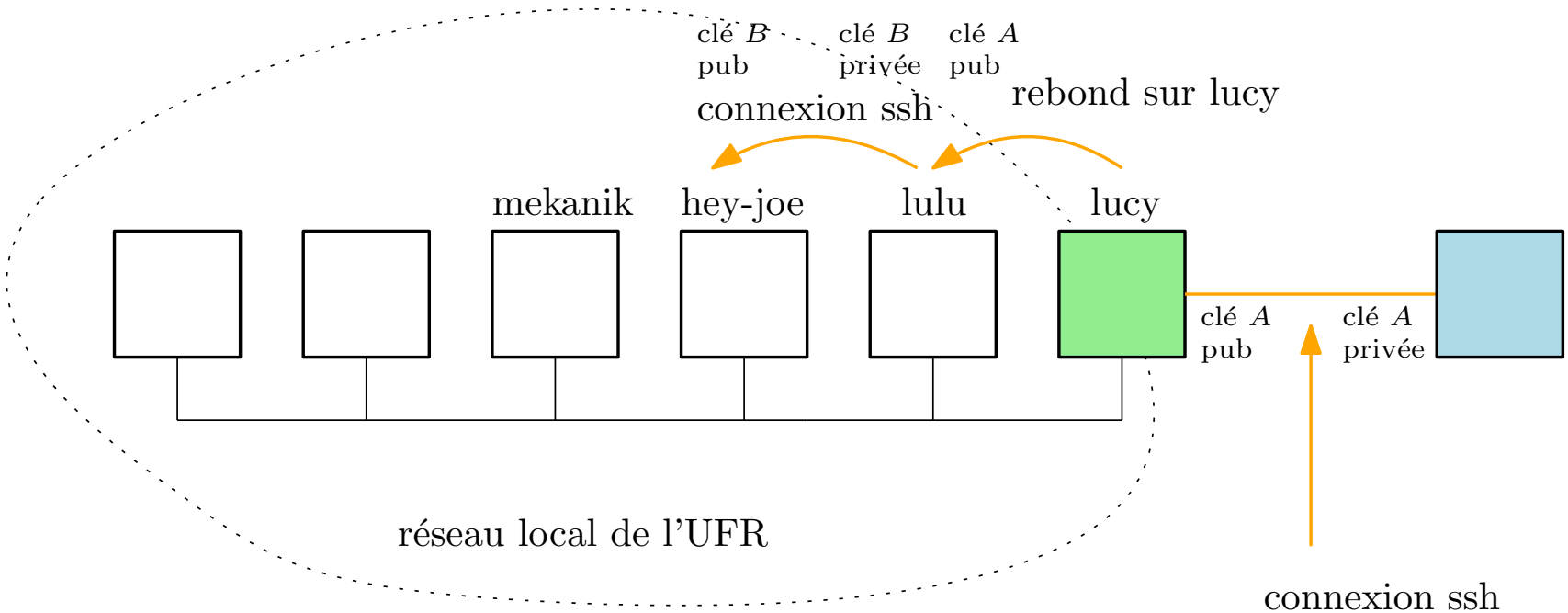
  ```
  lulu:~$ nc -l 7879          hey-joe:~$ nc lulu 7879
  ```

  Le serveur s'exécute sur lulu et le client sur hey-joe

Le réseau de l'UFR d'informatique

- Le réseau de l'UFR d'informatique est un réseau local accessible de l'extérieur uniquement par ssh sur la machine lucy. Cela signifie que :
  - on ne peut pas échanger directement des messages entre une machine extérieure et une machine interne au réseau de l'UFR
  - on ne peut donc pas joindre directement une machine du réseau interne de l'UFR depuis sa machine personnelle
  - depuis sa machine personnelle, on se connecte à lulu (sur le réseau interne) en effectuant une connexion ssh à lucy (accessible de l'extérieur) avec rebond sur lulu
  - on peut se connecter aux autres machines de l'UFR par ssh depuis lulu ou depuis sa machine avec rebond sur lucy
  - une fois connecté à la machine A, on travail localement sur A
  - pour ce cours, si vous voulez tester **deux applications** sur des machines distantes qui discutent entre elles, il faut que chaque application s'exécute sur une machine du **même réseau local**

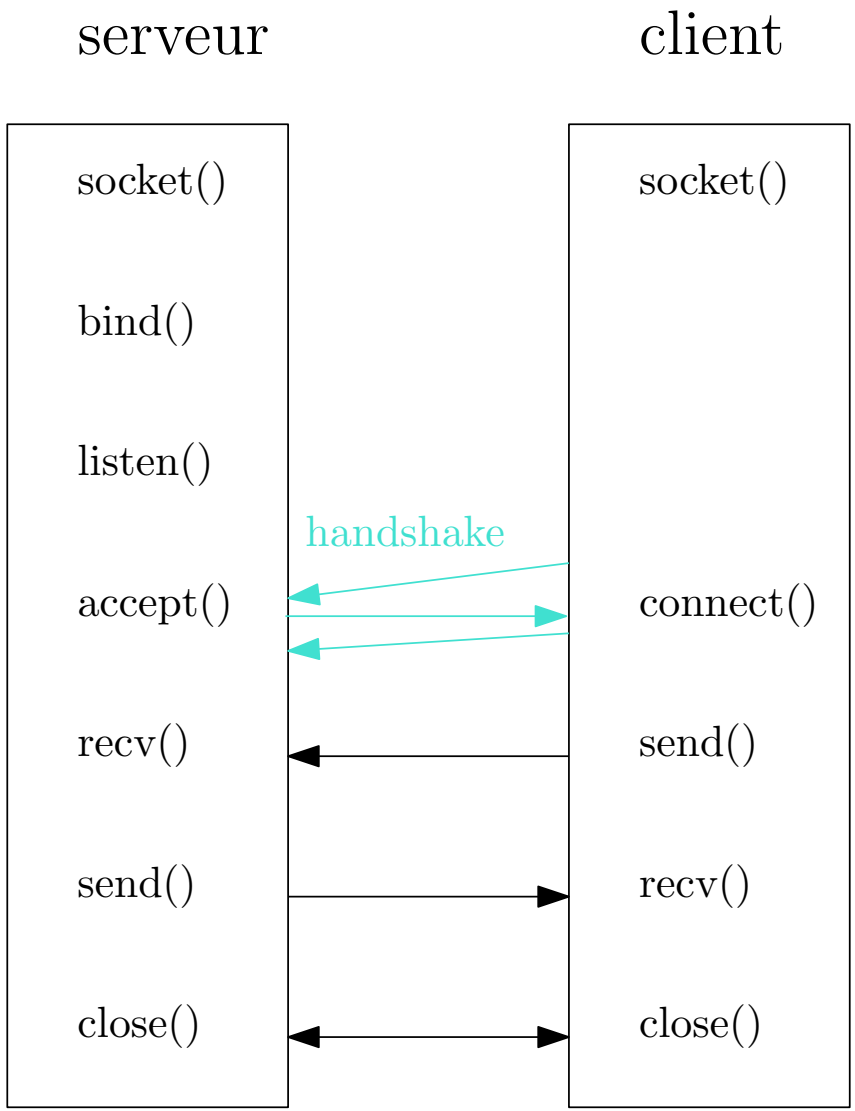
Le réseau de l'UFR d'informatique



Attention, depuis le réseau interne de l'UFR, lucy se nomme **lucette**

III - Serveur TCP

Communication TCP client/serveur



Un serveur TCP

Les étapes pour créer un serveur :

1. créer la *socket serveur*
2. préparer l'adresse avec port d'écoute du serveur
3. *lier* la socket serveur au port d'écoute
4. passer la socket en mode *écoute* (c'est-à-dire préparer la socket serveur à recevoir des connexions)
5. accepter une connexion et créer la *socket client*. La conversation peut commencer...
6. fermer la socket client à la fin de la conversation

Un serveur TCP IPv4

Socket et adresse

Le début est presque le même que pour le client

```
 //*** creation de la socket serveur ***
int sock = socket(PF_INET, SOCK_STREAM, 0);
if(sock < 0){ perror("creation socket"); exit(1);}

//*** creation de l'adresse du destinataire (serveur) ***
struct sockaddr_in adrsock;
memset(&adrsock, 0, sizeof(adrsock));
adrsock.sin_family = AF_INET;
adrsock.sin_port = htons(2121);
adrsock.sin_addr.s_addr = htonl(INADDR_ANY);
```

Pour un serveur, on veut en général pouvoir lier la socket à n'importe quelle interface disponible. On affecte donc la constante C **INADDR_ANY** en écriture *big-endian* au champ `sin_addr.s_addr` de l'adresse de socket.

Un serveur TCP IPv4

Lier la socket à un port d'écoute

Pour lier la socket à un numéro de port, on utilise la fonction C

```
int bind(int socket, const struct sockaddr *address, socklen_t address_len);
```

```
int r = bind(sock, (struct sockaddr *) &adrsock, sizeof(adrsock));
```

- Il faut faire un transtypage explicite du pointeur d'adresse vers une adresse de type `struct sockaddr *`.
- `r` vaut **0** en cas de succès et **-1** sinon avec `errno` positionné

  Ne pas oubliez de tester si la liaison a réussi ! ! !
- On peut également lier la socket à une IP ou une interface spécifique pour uniquement accepter les connexions sur cette adresse.

On a maintenant une socket serveur.

Un serveur TCP IPv4

Préparer la socket serveur à recevoir des connexions

Le serveur se déclare prêt à écouter les connexions sur le port en faisant appel à la fonction

```
int listen(int sockfd, int backlog);
```

- `backlog` est la taille maximale de la file de connexions en attente. `0` signifie sans limite, soit 4096, ou parfois 128 ! ! !
- `listen` retourne `0` en cas de succès et `-1` sinon avec `errno` positionné

```
int r = listen(sock, 0);
```

Un serveur TCP IPv4

Accepter une connexion

Pour accepter la demande de connexion d'un client, on a la fonction C

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

- Les paramètres `addr` et `addrlen` sont des pointeurs car ceux-ci vont être modifiés et récupérer respectivement l'adresse et port du client d'une part, et la taille réelle d'`addr` d'autre part. Ils peuvent également prendre la valeur **NULL** si on ne souhaite pas utiliser l'adresse de socket.
- Il faut faire un transtypage explicite du pointeur d'adresse vers une adresse de type `struct sockaddr *`.
- retourne le numéro du descripteur associé à la socket client en cas de succès et **-1** sinon avec `errno` positionné

Ne pas oubliez de tester si accept a réussi ! ! !

Un serveur TCP IPv4

Accepter une connexion

```
int sockclient = accept(sock, NULL, NULL);
if(sockclient == -1){
  perror("probleme socket client");
  exit(1);
}
```

Attention, il ne faut pas confondre le rôle des deux sockets créées :

- le descripteur de la socket de communication entre le serveur et le client est `sockclient`
- la socket du serveur, ici `sock`, écoute sur le port 2121 et attend avec l'appel à la fonction `accept`, les demandes de connexion des clients.

La conversation entre le serveur et le client peut commencer...

Un serveur TCP IPv4

Afficher l'adresse IPv4 et le port du client connecté

```
struct sockaddr_in adrclient;
memset(&adrclient, 0, sizeof(adrclient));
socklen_t size = sizeof(adrclient);

int sockclient = accept(sock, (struct sockaddr *) &adrclient, &size);
if(sockclient == -1) { perror("probleme socket client"); exit(1); }

char addr_buf[INET_ADDRSTRLEN];
memset(addr_buf, 0, sizeof(addr_buf));

if (inet_ntop(AF_INET, &(adrclient.sin_addr), addr_buf, sizeof(addr_buf)) == NULL)
  perror("erreur recuperation de l'adresse IPv4");
else
  printf("client connecte : %s %d\n", addr_buf, ntohs(adrclient.sin_port));
```

- On récupère l'adresse IPv4 du client dans `adrclient`.
- Puis, on récupère l'adresse IPv4 du client sous forme de chaîne de caractères avec la fonction `inet_ntop`.
- Il ne faut pas oublier de récupérer le numéro de port avec l'encodage de votre machine en faisant appel à la fonction `ntohs`.

Si le serveur souhaite terminer la communication dans un sens avec le client, il peut faire appel à la fonction :

```
int shutdown(int sockfd, int how)
```

où `sockfd` est la socket de connexion (avec le client) et `how` peut prendre une des valeurs suivantes :

- **SHUT_RD** ou **0** : fermeture de la socket pour la réception/lecture ⇒ les données pendantes sont lues, puis les `recv` suivants sont désactivés,
- **SHUT_WR** ou **1** : fermeture de la socket pour l'envoi/écriture ⇒ les données stockées dans le buffer d'envoi sont envoyées, puis les `send` suivants sont désactivés
- **SHUT_RDWR** ou **2** : fermeture en lecture et écriture de la socket.

Un serveur TCP IPv4

Fermeture partielle d'une socket

Au niveau du système, que se passe-t'il ?

Serveur		Client
Fermeture en écriture `shutdown(sockclient, SHUT_WR);` ⇒ envoi des données stockées dans le buffer d'écriture, puis envoi d'un message de fin qui termine définitivement l'envoi.	→	réception des données et du message de fin qui provoque la fermeture en lecture sur la socket du client ⇒ `recv` retourne la valeur **0**.
Fermeture en lecture `shutdown(sockclient, SHUT_RD);` ⇒ la socket de communication avec le client est fermée en lecture après que toutes les données stockées dans le buffer de lecture (données pendantes) ont été transmises à l'application. Puis envoi d'un message de fin qui termine définitivement la lecture.	→	réception par le système du message de fin qui provoque la fermeture en écriture sur la socket du client ⇒ le signal **SIGPIPE** est envoyé par le système et `send` retourne la valeur **-1** avec l'erreur « connection reset by peer ».

A la réception du signal **SIGPIPE**, le client s'arrête sauf si

solution 1 celui-ci ignore le signal.

```
struct sigaction action = {0};
action.sa_handler = SIG_IGN;
sigaction(SIGPIPE, &action, NULL);

...

int ecrit = send(fdsock, buf, strlen(buf), 0);
if(ecrit < 0) {
  perror("erreur ecriture");
  ...
 }
```

A la réception du signal **SIGPIPE**, le client s'arrête sauf si

solution 2 le flag **MSG_NOSIGNAL** est passé à `send`.

```
int ecrit = send(fdsock, buf, strlen(buf), MSG_NOSIGNAL);
if(ecrit < 0) {
  perror("erreur ecriture");
  ...
 }
```

Dans le cas où le pair distant a clos la socket en lecture, le flag **MSG_NOSIGNAL** passé à `send` empêche l'envoi du signal **SIGPIPE** .

L'appel à `shutdown` ne libère pas le descripteur de socket. Il faut invoquer `close` pour cela.

Un appel à `shutdown` est définitif. On ne peut réouvrir la lecture ou l'écriture sur la socket.

Ce mécanisme permet de fermer proprement un ou les deux côtés de la socket. Il peut être utile, par exemple, si on veut prévenir le pair distant que l'envoi est terminé mais que l'on veut pouvoir continuer à recevoir des données.

Un serveur TCP IPv6

Qu'est-ce qui change ?

```
int sock = socket(PF_INET6, SOCK_STREAM, 0);
if(sock < 0){
  perror("creation socket");
  exit(1);
}

struct sockaddr_in6 address_sock;
memset(&address_sock, 0, sizeof(address_sock));
address_sock.sin6_family = AF_INET6;
address_sock.sin6_port = htons(2121);
address_sock.sin6_addr = in6addr_any;
```

`in6addr_any` est une variable de type `struct in6_addr` qui contient l'adresse locale au format IPv6 avec octets déjà dans l'ordre réseau,

Un serveur TCP IPv6

Qu'est-ce qui change ?

La constante **IN6ADDR_ANY_INIT** peut également être utilisée mais **uniquement à l'initialisation** d'une variable de type `struct in6_addr`.

```
int sock = socket(PF_INET6, SOCK_STREAM, 0);
if(sock < 0){
  perror("creation socket");
  exit(1);
}

struct sockaddr_in6 adrsock = {.sin6_family = AF_INET6,
          .sin6_port = htons(atoi(args[1])), .sin6_flowinfo = 0,
          .sin6_addr = IN6ADDR_ANY_INIT, .sin6_scope_id = 0};

/* memset(&adrsock, 0, sizeof(adrsock));
   adrsock.sin6_family = AF_INET6;
   adrsock.sin6_port = htons(2121);
   adrsock.sin6_addr = in6addr_any; */
```

Un serveur TCP IPv6

Qu'est-ce qui change ?

- Le paramètre de type `struct sockaddr*` de `bind` et d'`accept` doit en fait être de type réel `struct sockaddr_in6*`.
- Les étapes `bind`, `listen` et `accept` sont alors les mêmes qu'en IPv4.

Si on veut de plus afficher l'adresse IP et le port du client

```
struct sockaddr_in6 adrclient;
socklen_t size=sizeof(adrclient);

int sockclient = accept(sock, (struct sockaddr *) &adrclient, &size);
if (sockclient >= 0) {
    char addr_buf[INET6_ADDRSTRLEN];
    inet_ntop(AF_INET6, &(adrclient.sin6_addr), adr_buf, sizeof(adr_buf));
    printf("client connecte : %s %d\n", addr_buf, adrclient.sin6_port);
}
```

IV - La lecture sur socket

La lecture sur socket

TCP est un protocole par *flux*.

$\Rightarrow$ on ne peut considérer que les données envoyées par un `send` pourront être entièrement lues par un unique `recv`.

Si deux applications A1 et A2 communiquent et

- A1 envoie 100 octets à A2 avec un unique `send`,
- alors on ne peut pas savoir avec combien de `recv`, A2 pourra lire les 100 octets.

$\Rightarrow$ on ne peut considérer que les données envoyées par k `send` pourront être entièrement lues par k `recv` lisant la quantité de données envoyée par le send correspondant.

Si deux applications A1 et A2 communiquent et

- A1 envoie avec 3 `send`, 100 octets par `send` à A2,
- alors on ne peut pas savoir avec combien de `recv`, A2 pourra lire les 300 octets.

La lecture sur socket

C'est pourquoi on a besoin de protocoles très précis décrivant les formats des messages échangés.

- Si l'application sait exactement combien d'octets elle doit lire ⇒ boucle sur le `recv` tant que tous les octets n'ont pas été lus.
- Si l'application sait par quelle suite de caractères termine le message qu'elle doit lire ⇒ boucle sur le `recv` tant que la suite de caractères n'a pas été trouvée.
  Attention si par exemple le message termine par les deux caractères `\r\n`, alors il est possible que ces deux caractères ne soient pas récupérés par le même `recv`. Le message d'un premier `recv` peut terminer par `\r` et le message suivant commencer par `\n`.
- Si l'application sait que l'autre application fermera la connexion juste après l'envoi, elle peut faire une boucle sur `recv` tant que ce dernier ne retourne pas **0**.

La lecture sur socket

→ Pour le cas où une suite de caractères annoncent la fin de la transmission, il faut pouvoir rechercher cette suite dans le buffer.

→ Il peut être nécessaire d'accumuler dans un buffer les messages reçus (ou une partie des messages reçus) jusqu'à obtenir le bon nombre d'octets ou la suite de caractères attendue ou un retour nul de `recv`.

Pour cela, vous avez à votre disposition les fonctions :

- `str...` : si le buffer est une chaîne de caractères (termine par le caractère `\0`).
  voir `strchr`, `strstr`, `strcpy`...

- `mem...`
  voir `memchr`, `memmem` (non POSIX), `memmove`...

La lecture sur socket

Situations de blocage

En mode connecté, des *situations de blocage* peuvent arriver dès qu'une des applications communicantes ne respecte pas le protocole.

Si l'application A1 est en attente d'un message d'une application A2 et que cette dernière n'envoie pas la totalité du message attendu, alors A1 peut bloquer indéfiniment.

Pour un serveur, cela peut être problématique car cela permet des *attaques*.