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**.

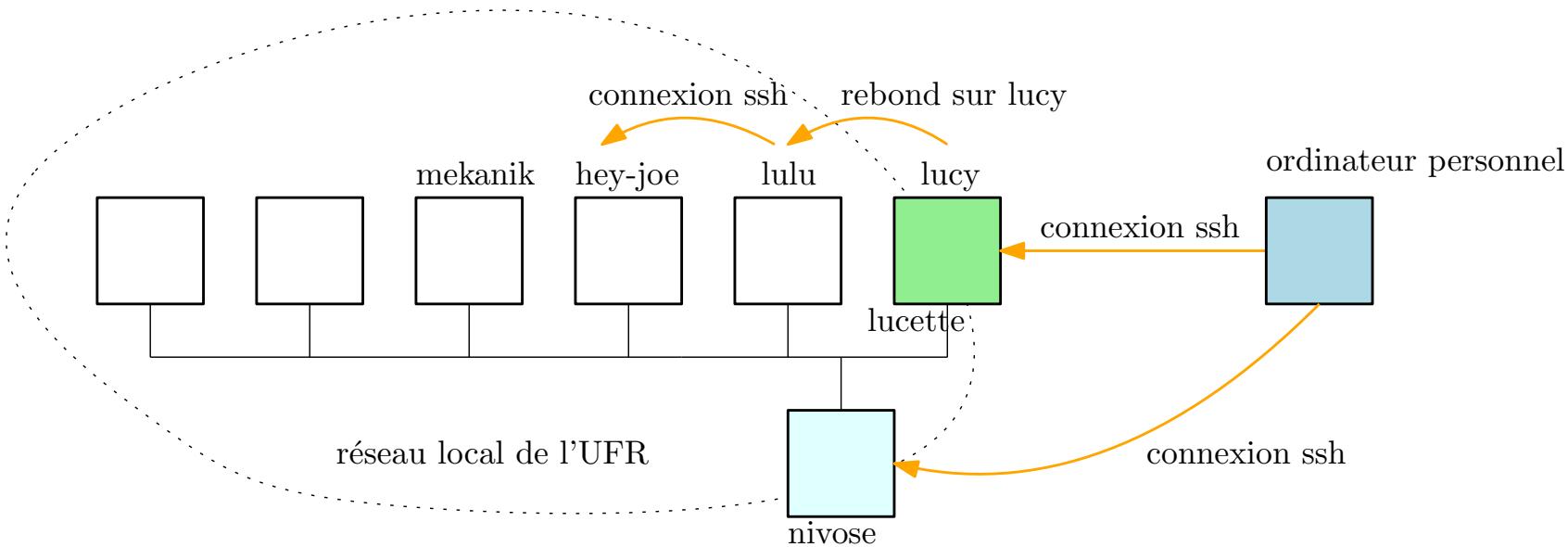
  ```
  lulu:~$ nc -l 7879          hey-joe:~$ nc lulu 7879
  ```

  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**

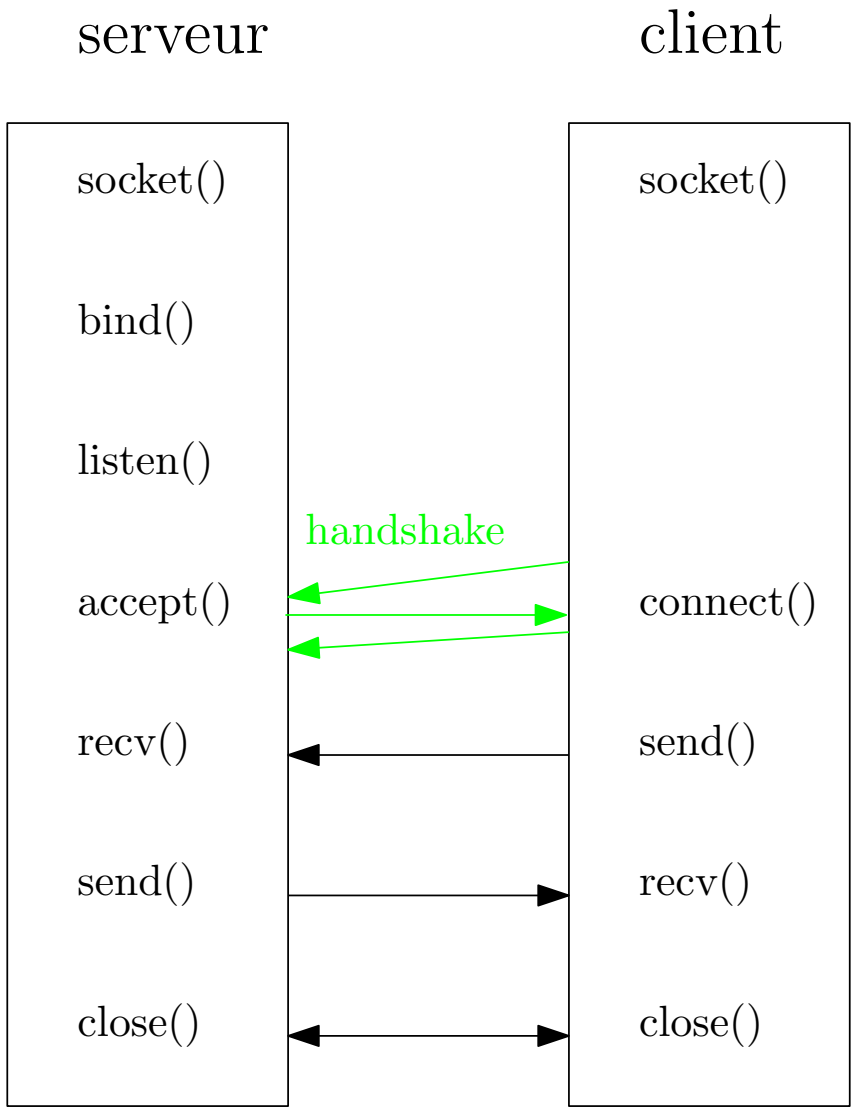
Le réseau de l'UFR d'informatique



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

III - Serveur TCP

Communication TCP client/serveur



Communication TCP client/serveur

Retour sur `struct sockaddr`

- `struct sockaddr_in` a la même taille que `struct sockaddr`
- `struct sockaddr_in6` a une taille plus grande que `struct sockaddr`

Les fonctions comme `connect` ou `accept` (que nous allons voir aujourd'hui), doivent fonctionner pour des adresses IPv4 ou IPv6. Le 2ème paramètre est donc de type `struct sockaddr` et le 3ème paramètre contient sa taille réelle.

```
int r = connect(sock, (struct sockaddr *) &adrso, sizeof(adrso));
```

`connect` connaît l'adresse mémoire où est stockée l'adresse de socket et connaît également la taille réelle de cette dernière $\Rightarrow$ il a toutes les informations nécessaires pour récupérer l'adresse de connexion.

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. 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 *) &adrso, sizeof(adrso));
```

- 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 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 t 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 *`.
- `r` vaut 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 ! ! !

```
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 les connexions des clients.

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

Un serveur TCP IPv4

Afficher l'adresse 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));

inet_ntop(AF_INET, &(adrclient.sin_addr), addr_buf, sizeof(addr_buf));
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`.

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 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 = {AF_INET6,  htons(2121),
                               0, IN6ADDR_ANY_INIT, 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 ?

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

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 - Connexion par adresse de noms

Connexion par adresse de noms

Pour l'humain, retenir une adresse composée de noms plutôt qu'une suite de nombres est plus simple.

Si on souhaite permettre à l'utilisateur d'entrer une adresse de noms,

- on ne peut plus utiliser les fonctions `inet_pton` et `inet_ntoa`,
- il faut donc pouvoir traduire ces adresses de nom en adresse IP :
  - On a vu que cela signifie qu'il faut interroger un dictionnaire (en général service DNS) pour obtenir la traduction d'une adresse de noms en adresse IP.
  - On peut vouloir ne retenir qu'un sous-ensemble d'adresses répondant à des critères particuliers (IPv4, IPv6, avec socket TCP ou UDP)
  - Mais comment fait-on en C ?

Connexion par adresse de noms

struct addrinfo

On commence par remplir une variable `hints` de type `struct addrinfo` avec des indications comme le type d'adresses ou de socket voulu.

```
struct addrinfo {
  int ai_flags;              /* options */
  int ai_family;             /* famille de protocols pour la socket */
  int ai_socktype;           /* type de socket */
  int ai_protocol;           /* protocole pour la socket */
  socklen_t ai_addrlen;      /* taille de l'adresse */
  struct sockaddr *ai_addr; /* addresse: IP, port... */
  char *ai_canonname;        /* nom canonique de l'adresse */
  struct addrinfo *ai_next; /* pointeur sur le suivant dans la liste */
};
```

Connexion par adresse de noms

struct addrinfo

Valeurs d'initialisation de `hints` pour obtenir des adresses IPv4 pour socket TCP :

- `ai_flags` : égal à `0` pour le moment
- `ai_family` : égal à `AF_INET`
- `ai_socktype` : égal à `SOCK_STREAM`
- `ai_protocol` : égal à `0` dans le cas de TCP
- `ai_canonname` : égal à `0` pour le moment

Tous les autres éléments de `struct addrinfo` passés via `hints` doivent être mis à `0` (ou au pointeur `NULL`).

Connexion par adresse de noms

Initialisation de `hints`

On dit que l'on veut récupérer des adresses IPv4 pour socket TCP

```
struct addrinfo hints
memset(&hints, 0, sizeof(hints));

hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
```

Et maintenant, comment interroger le dictionnaire pour traduire les noms d'adresses ?

Connexion par adresse de noms

getaddrinfo

On doit utiliser la fonction

```
int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

- `node` : chaîne de caractères contenant l'adresse de noms
- `service` : chaîne de caractères contenant le numéro de port
- `hints` : contient des indications sur le type d'adresses, de socket...
- `res` : liste chaînée de pointeurs de `struct addrinfo` contenant les différentes adresses de socket possibles