Programmation répartie avec les sockets

Patrice Torguet

IRIT

Université Paul Sabatier

Plan

- Problèmes liés au réseau
- Protocoles applicatifs
- Sockets BSD
- Protocoles de Transport
- Programmation en C/C++
- Spécificités de Windows
- Conclusion

Caractéristiques des réseaux

- Latence
- Débit
- Fiabilité

Latence

- Durée nécessaire pour envoyer un bit de donnée d'un endroit à un autre dans le réseau (latency, lag, ping …)
- La latence diminue l'interactivité d'une application distribuée
- Raisons :
  - Délai du à la vitesse de la lumière (8.25 ms par fuseau horaire)
  - Délai ajouté par les ordinateurs (traitements)
  - Délai ajouté par le réseau (gestion des erreurs, de la congestion, commutation, traduction données en signal…)
- Autre notion : la gigue (variation de la latence)

Débit

- Nombre de bits qui peuvent être acheminés par le réseau en 1 seconde (throughput, bande passante, bandwidth)
- Dépends des types de câbles (ou du support de propagation) et des équipements réseaux

Fiabilité

- Un réseau peut perdre/détruire des données (congestion)
  - Retransmissions possibles par Transport
- Les données peuvent être modifiées (erreurs de transmission)
  - Retransmissions ou destruction (cf. plus haut)
- Si on veut de la fiabilité il faut gérer des acquittements (soit au niveau Transport soit au dessus - application en général)

Protocoles applicatifs

- Règles utilisées par deux applications pour communiquer
- Au niveau application :
  - Format des messages
  - Sémantique d'envoi/réception
  - Conduite à tenir en cas d'erreur
- cf. RFC proto app : HTTP, FTP, SMTP, POP...

Format des messages

- Indique ce que contiennent les messages échangés
- Côté émission : que doit on y mettre
- Côté réception : comment doit-on décortiquer un message reçu

Types de messages

- En général un protocole gère plusieurs types de messages
- Contrôle :
  - Connexion / Déconnexion / Acquittement / Synchronisation ...
- Information :
  - Ordres / Données / Codes d'erreurs...

Sémantique d'envoi/réception

- L'émetteur et le récepteur doivent se mettre d'accord sur ce que le récepteur peut déduire quand il reçoit un certain message
- Quelles actions le récepteur doit exécuter lors de la réception d'un certain message

Conduite à tenir en cas d'erreur

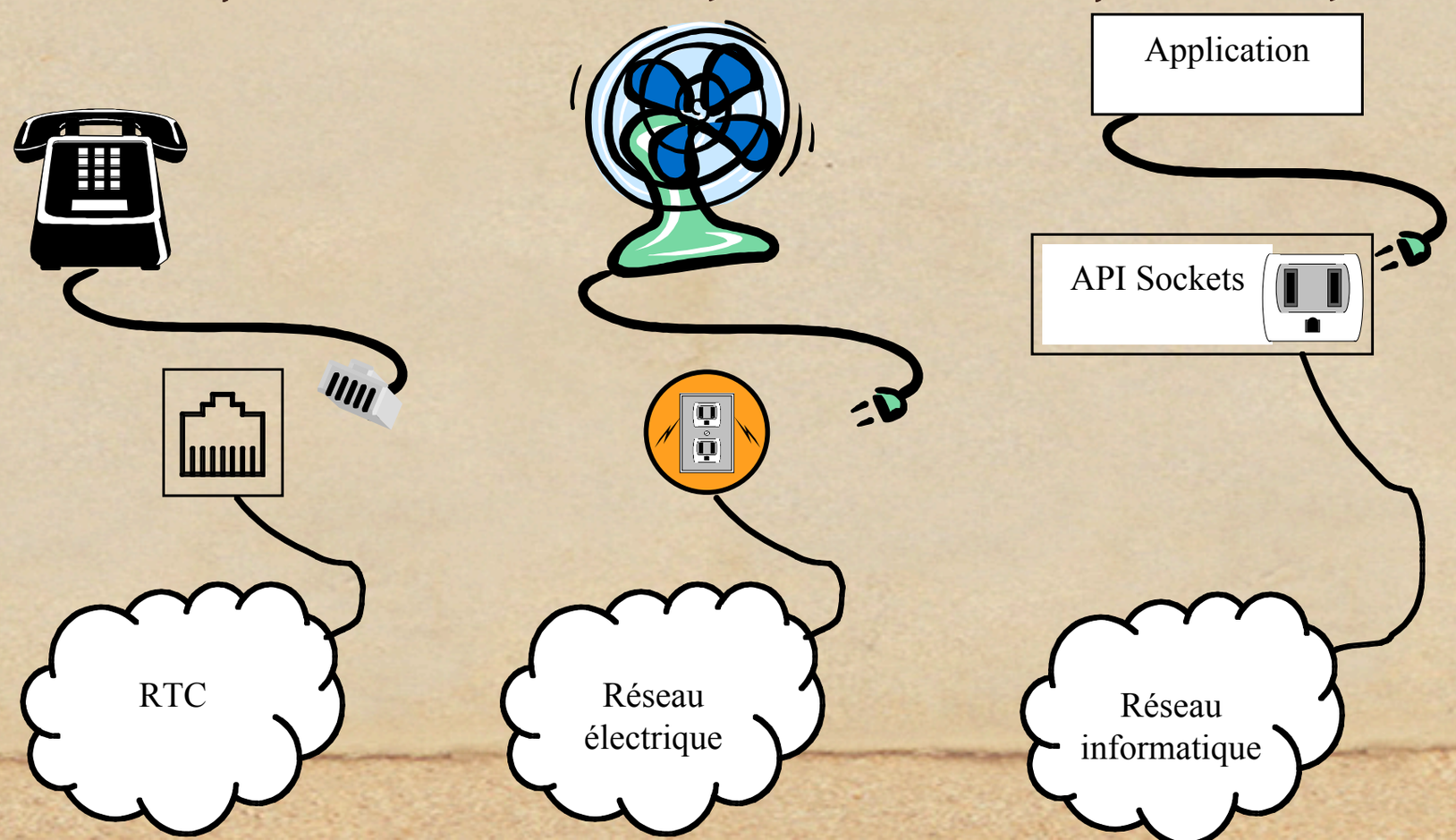
- Ensemble de scénarios d'erreur et ce que chaque application doit faire pour chaque cas

Les sockets BSD

- BSD (Berkeley Software Distribution) est un système d'exploitation de type UNIX développé par l'université de Berkeley depuis 1977
- Parmi les avancées introduites par ce système d'exploitation on trouve les sockets (version 4.3 de BSD - 1983)
- Le concept a été depuis repris par tous les systèmes d'exploitations

Sockets BSD

- Une socket est une abstraction correspondant à une extrémité d'un canal de communication
- Le nom socket (prise) vient d'une analogie avec les prises électriques et téléphoniques



Sockets BSD

- Les sockets sont aussi une API (interface de programmation) pour :
  - manipuler des informations liées à la communication (adresses source et destination)
  - créer un canal de communication si nécessaire
  - envoyer et recevoir des messages protocolaires applicatifs
  - contrôler et paramétrer la communication

Sockets BSD

- Les sockets BSD permettent des communications
  - internes à un systèmes d'exploitation UNIX (domaine AF_UNIX)
  - à distance
    - en utilisant TCP/IP (domaine AF_INET)
    - ou d'autres piles de protocoles (ATM notamment)

Sockets BSD

- L'abstraction peut être
  - orientée réseau
    - pour TCP/IP par exemple une socket correspond à un quintuplet : @IP locale, port local, @IP distante, port distant, protocole de transport (TCP ou UDP)
  - orienté programmation
    - une socket est assimilée à un descripteur de fichier (comme les FIFO - pipes)

Protocoles de Transport

- Dans le domaine AF_INET on peut communiquer
  - en mode connecté (STREAM)
    - on utilise des connections TCP
  - en mode non connecté (DGRAM)
    - on utilise des datagrammes UDP indépendants
    - point à point ou multipoint (broadcast / multicast)
  - en mode “raw” (pour accéder aux couches basses - ICMP par exemple)

Protocoles de Transport

- Notion de port
  - Sur un même ordinateur plusieurs applications peuvent communiquer en même temps
  - Problème : comment savoir à quelle application on veut parler
  - Solution : chaque application est identifiée par un numéro unique (pour une machine donnée et pour un protocole donné) appelé numéro de port (entier sur 16 bits - 65535 ports différents - 0 n'est pas utilisé)

Protocoles de Transport

- Différents types de ports
  - ports systèmes ou bien connus (1-1023) - réservés au système d'exploitation (exemple 80 - serveur web)
  - ports utilisateurs ou enregistrés (1024- 49152) - ne doivent être utilisé que par des applications spécifiques (comme les premiers) enregistrées auprès de l'organisme IANA (Internet Assigned Numbers Authority - www.iana.org) (exemple 26000 - Quake)
  - ports privés ou dynamiques (les autres) - utilisés par les applications non encore enregistrées et par les clients TCP

IP : Internet Protocol

- Rappels :
  - gère l'adressage (@IP) et le routage dans Internet
  - gère la fragmentation pour satisfaire la taille max des trames des réseaux traversés : MTU
    - Attention : augmente la probabilité de perte (lors de la reconstruction si un seul fragment est perdu, le datagramme est détruit)
  - gère aussi le TTL : nombre max de routeurs que le datagramme peut traverser

Sockets STREAM / TCP

- Rappels sur TCP :
- offre un flot bidirectionnel d'octets entre 2 processus
  - Fiable (ni perte, ni duplication) et ordonné
  - "Connexion virtuelle" entre les 2 locuteurs (on peut donc détecter les ruptures de connexion)
  - Le plus utilisé aujourd'hui (mail, web, ftp...)
- 3 phases à programmer : connexion, dialogue, déconnexion

Sockets DGRAM / UDP

- Rappels sur UDP:
- Les données sont transférées dans des datagrammes UDP échangés indépendamment les uns des autres
  - Non fiable (pertes et duplications possibles) et non ordonné : best effort
  - Plus rapide que TCP
  - Utilisé par les applications multimédia (audio, vidéo, jeux)
- Envoi/réception de messages en utilisant un socket

Sockets DGRAM / UDP

- Avantages
- Protocole plus simple (pas de gestion des connexions virtuelles, pas de gestion de la fiabilité) et donc moins coûteux pour chaque machine
- Protocole un tout petit peu plus rapide (pas de gestion de l'ordre et d'évitement de la congestion) : les messages sont envoyés directement (pas besoin d'attendre lorsque la fenêtre de réception est pleine) et donnés à l'application dès qu'ils sont reçus (pas de remise en ordre)

Sockets DGRAM / UDP

- Inconvénients
- Problèmes de fiabilité (perte de messages dus essentiellement à la congestion) : si on veut de la fiabilité il faut la gérer au niveau application

Sockets DGRAM / UDP

- Diffusion totale et restreinte
- Pour éviter d'envoyer le même message vers plusieurs destinations on peut utiliser la diffusion avec UDP
- Diffusion totale : vers toutes les machines d'un réseau local (@ IPv4 de broadcast : id réseau + 255[.255.255] ou 255.255.255.255)
  - Utile pour les applications restreintes à un réseau local mais pas généralisable sur Internet
  - Nécessite des traitements par toutes les machines du réseau local : coûteux

Sockets DGRAM / UDP

- Diffusion restreinte : vers un ensemble de machine qui peut varier dynamiquement
  - Les @IPv4 multicast : de 224.0.0.0 à 239.0.0.0
    - 224.* sont réservées
    - 239.* sont réservées ou ne sont pas routées à l'extérieur d'un réseau local
    - 225.0.0.0 à 238.255.255.255 pour une utilisation sur Internet
  - En IPv6 toutes les adresses commençant par FF.
    - ffXe::/16 sont utilisables sur Internet. Quelque soit X.
    - ffX5::/16 sont restreintes au réseau local. (Pour les tests en local).

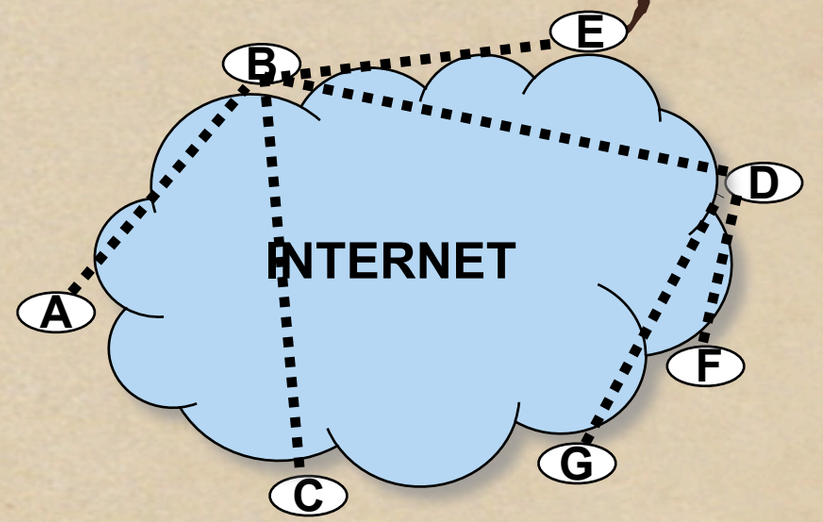
Sockets DGRAM / UDP

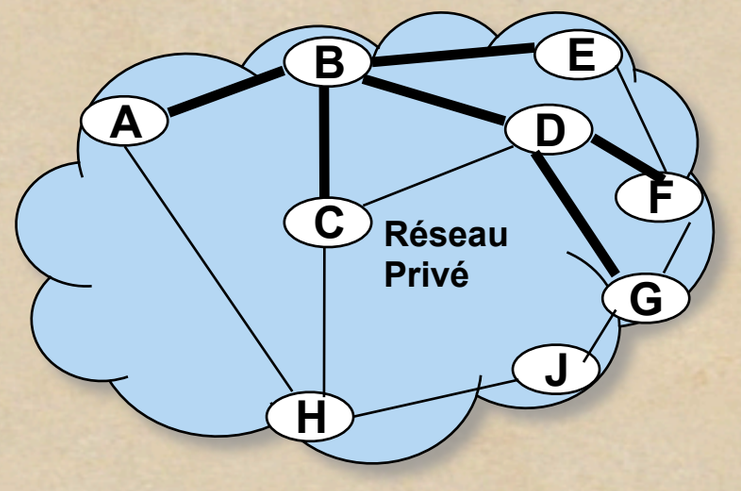
- Diffusion restreinte : principe
  - Une application sur une machine s'inscrit à une @ IP multicast
  - Les routeurs propagent les inscriptions et créent un arbre de diffusion (protocole IGMP)
  - Quand une application envoie un message vers une @ IP multicast, l'arbre de diffusion est utilisé et toutes les machines reçoivent le message (dans la limite du TTL utilisé)

Sockets DGRAM / UDP

- Diffusion restreinte : limitations
  - La plupart des routeurs le gèrent mais la plupart des administrateurs réseau le limitent au réseau local
  - Les FAI le réservent à la télé sur internet
  - Solution : création de tunnels entre les réseaux locaux (le tunnel encapsule les datagrammes à destination d'@ IP mcast dans des datagrammes normaux voire même dans des segments TCP)
    - Overlay multicast et application level multicast

Overlay multicast





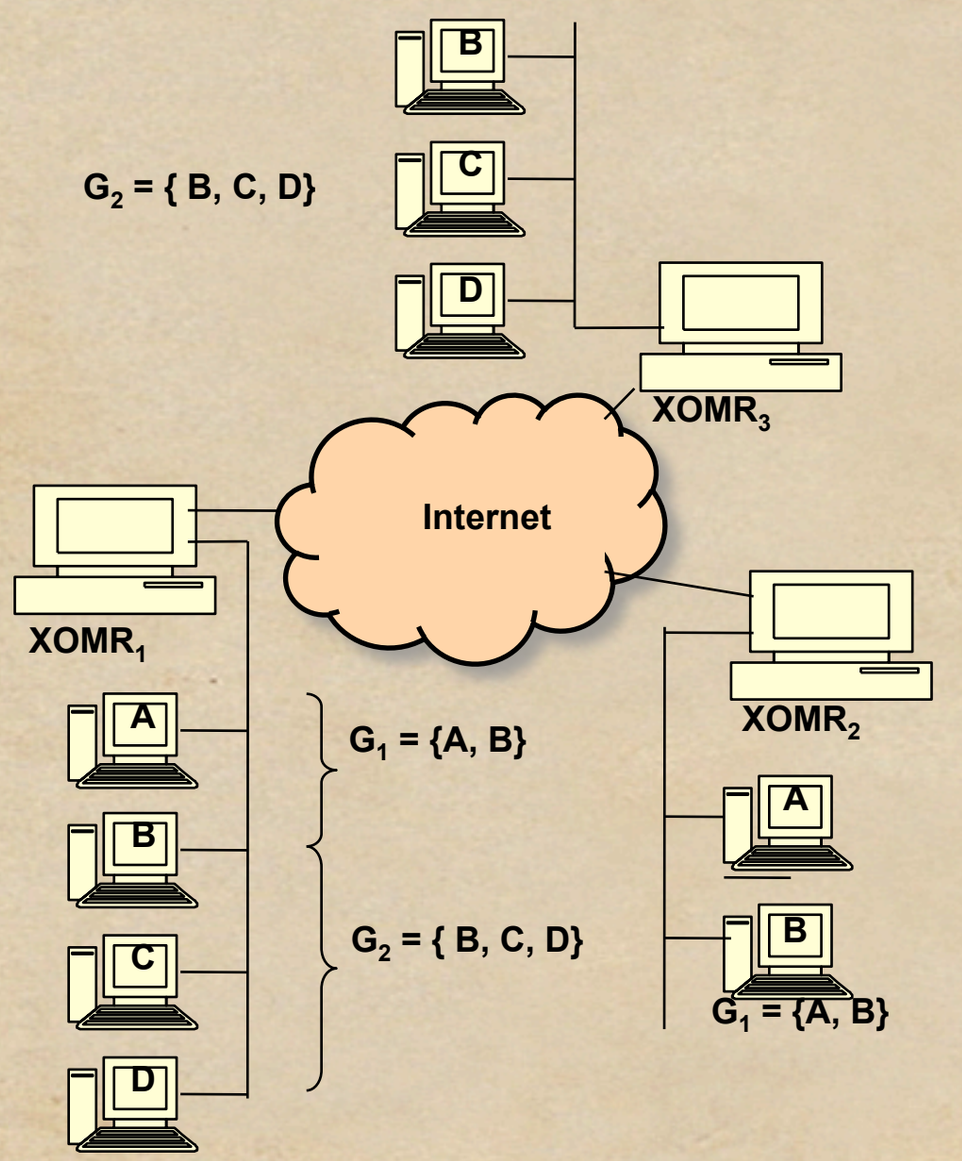
"Overlay" Multicast :

- Plusieurs vers plusieurs
  - Plusieurs émetteurs pour un même groupe
  - Arbres basés sur les sources
- Réseau Ouvert
  (indépendant des domaines de gestion)
- Peut être adapté à une application
  - Considérations de QoS de bout en bout
- Gestion des groupes efficaces

IP Multicast Standard :

- En général : 1 vers plusieurs
  - Un seul émetteur
  - Arbre basé sur une racine
- Réseau Privé
  (ne fonctionne que sur un seul domaine de gestion)
- Indépendant de l'application

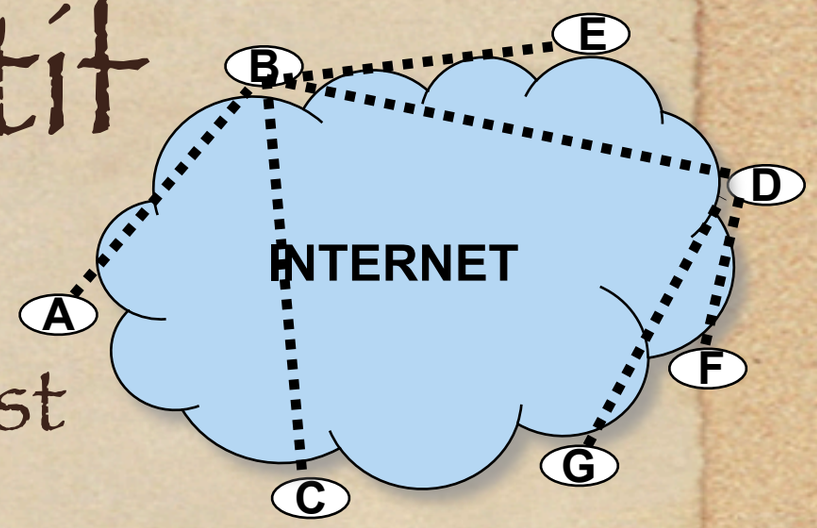
Overlay multicast : exemple XOM



Communications point à point (UDP ou TCP) entre les routeurs d'overlay (XOMR)

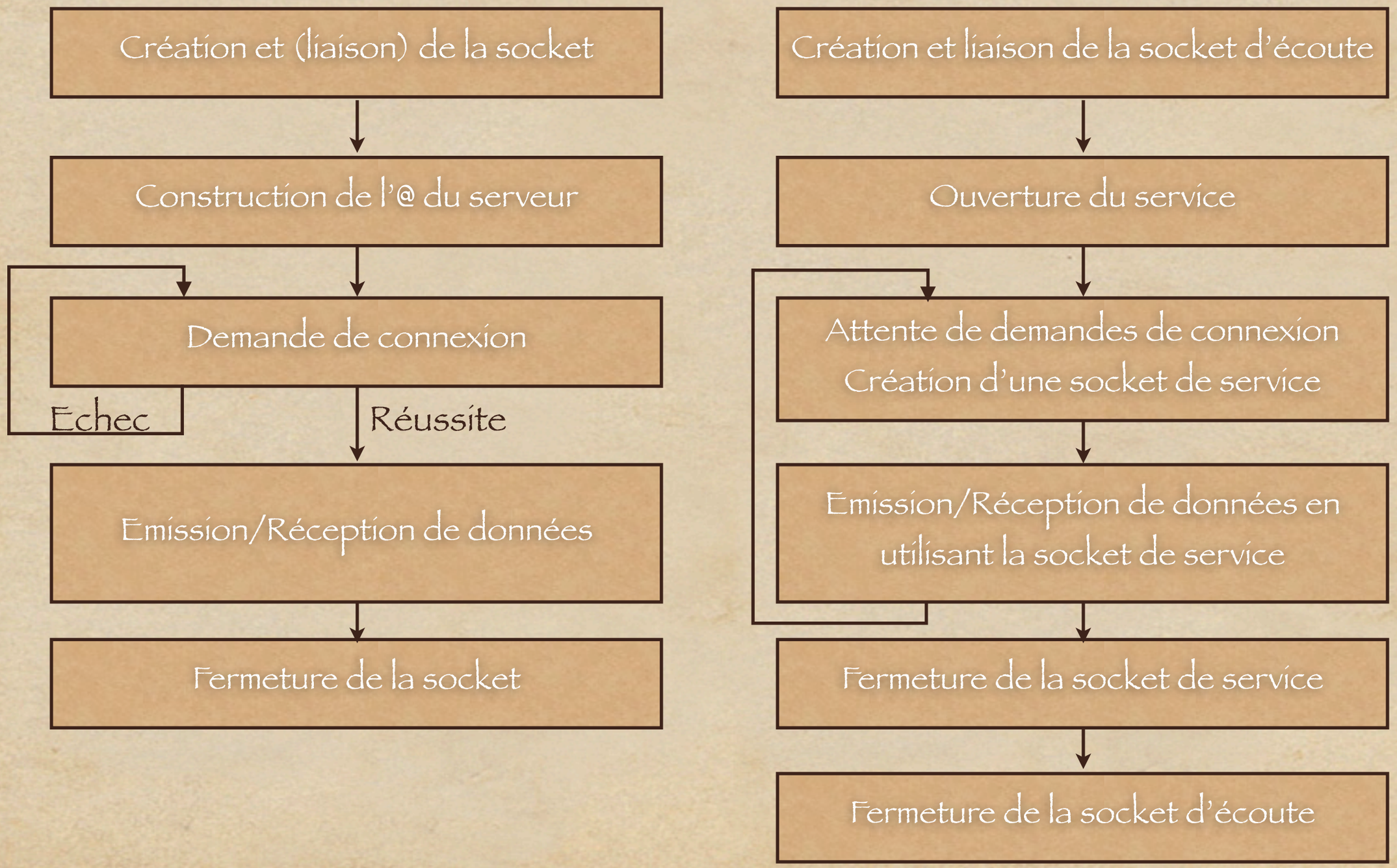
Multicast standard sur les LANs

Multicast applicatif

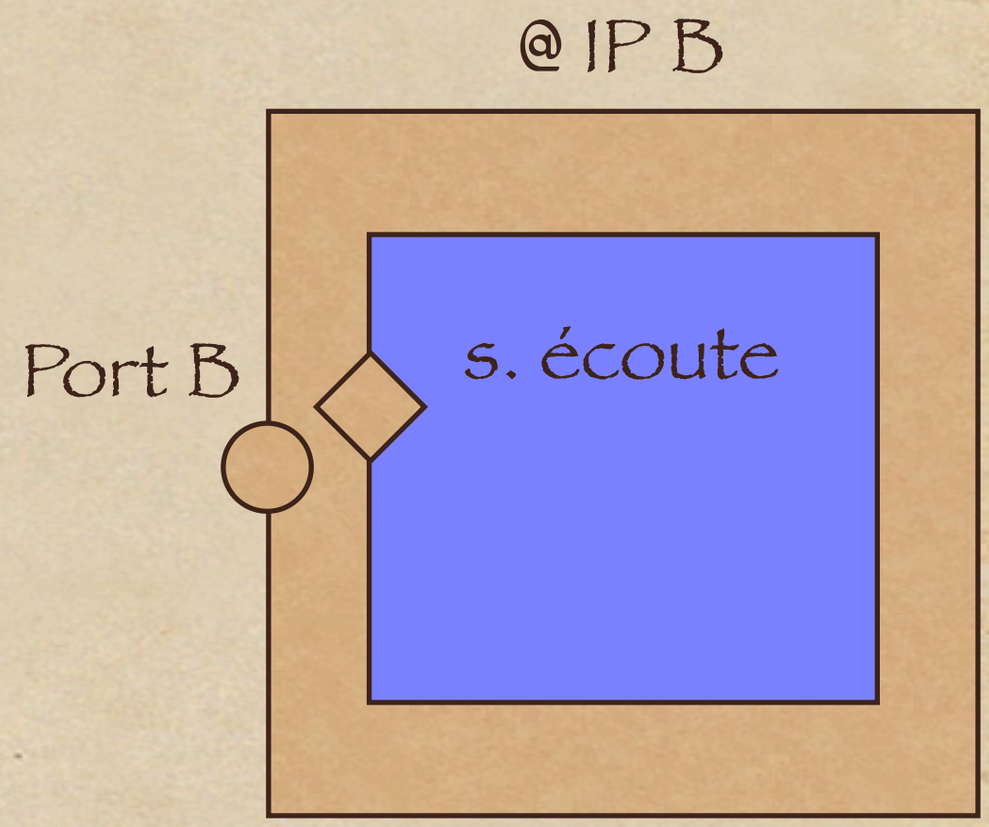


- Même principe que l'overlay mcast
- Mais l'overlay est géré par l'application (pas de "routeurs" d'overlay : A,B...G sont les applis)
- Avantage : on fait exactement ce que l'on veut (le routage peut être intelligent)
- Inconvénient : augmente la complexité de l'application

TCP et le Modèle client/serveur



Création sock écoute + bind



s. écoute

@ loc = @ IP B ou Any

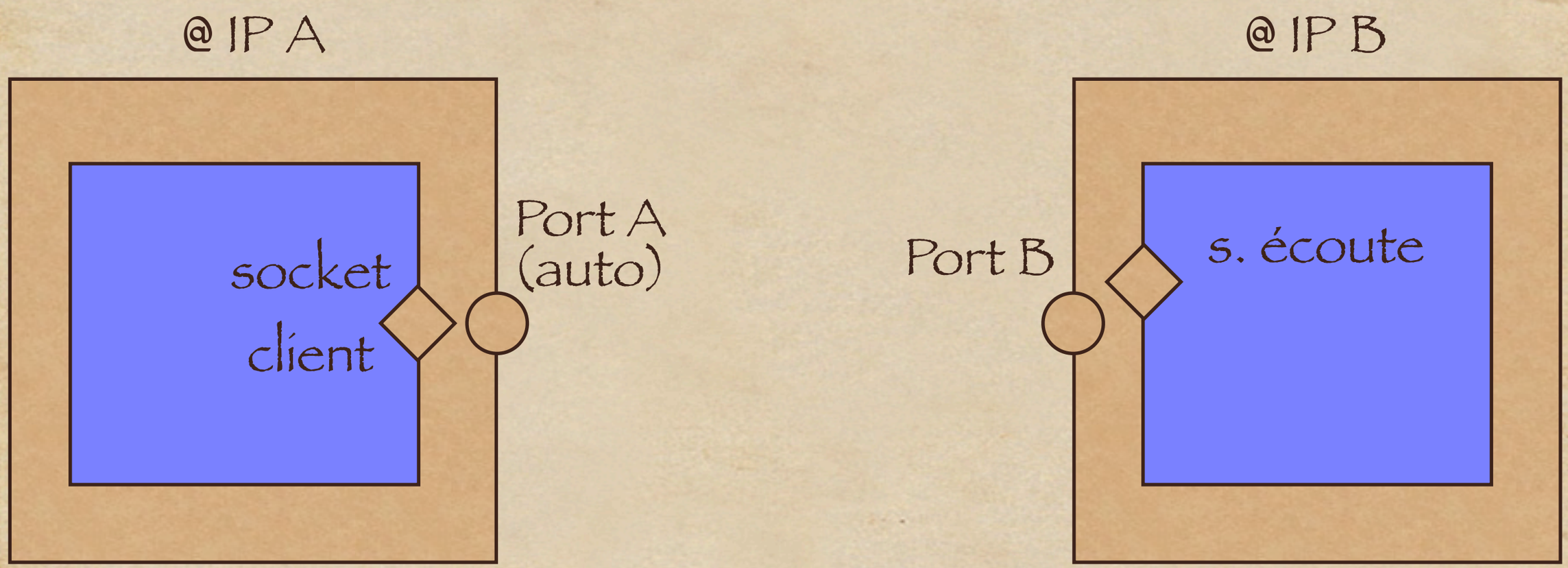
port loc = port B

@ dist = Any

port dist = 0

proto = TCP

Création sock client + bind



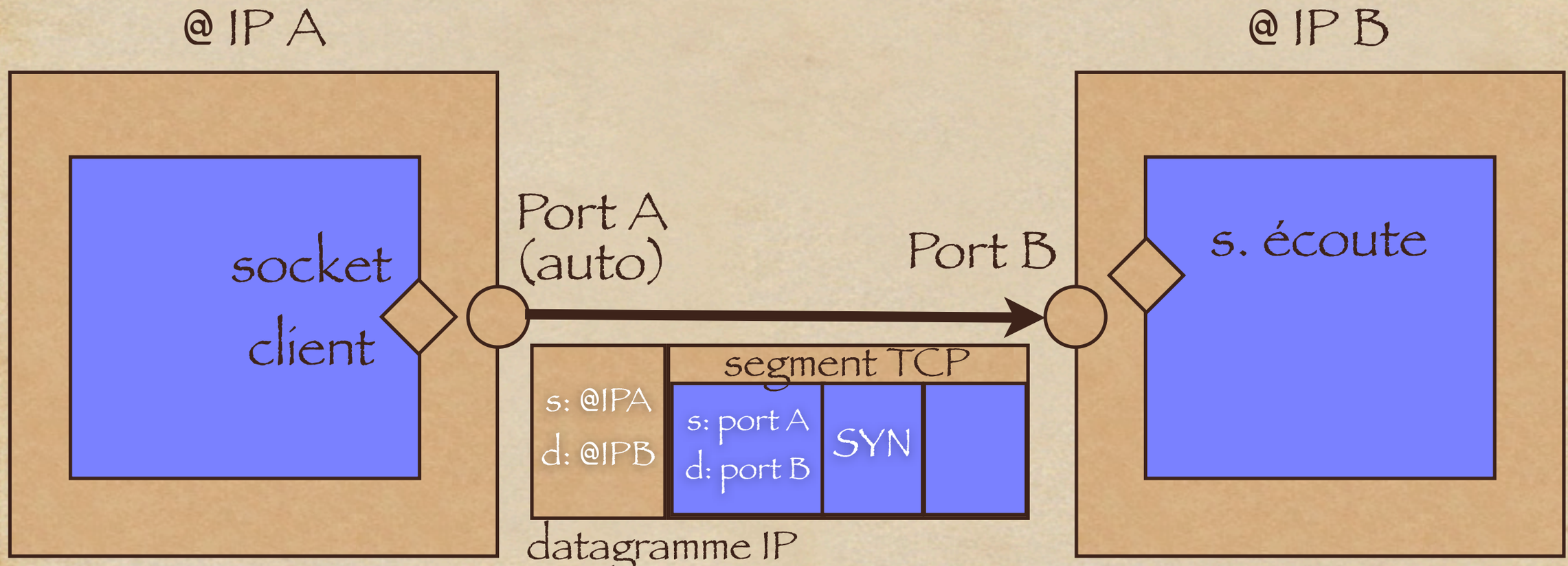
socket client

@ loc = @ IP A
port loc = port A
@ dist = Any
port dist = 0
proto = TCP

s. écoute

@ loc = @ IP B ou Any
port loc = port B
@ dist = Any
port dist = 0
proto = TCP

Demande de Connexion



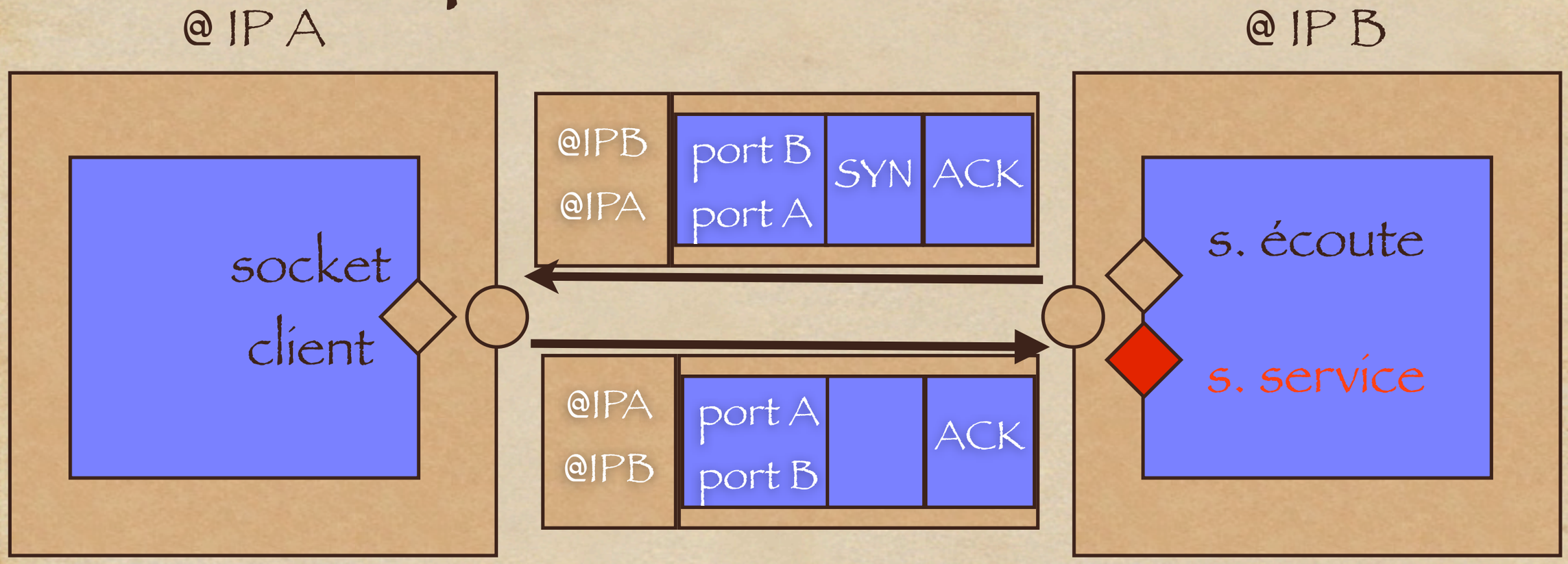
socket client

@ loc = @ IP A
port loc = port A
@ dist = @ IP B
port dist = port B
proto = TCP

s. écoute

@ loc = @ IP B ou Any
port loc = port B
@ dist = Any
port dist = 0
proto = TCP

Acceptation de connexion



socket client

@ loc = @ IP A
port loc = port A
@ dist = @ IP B
port dist = port B
proto = TCP

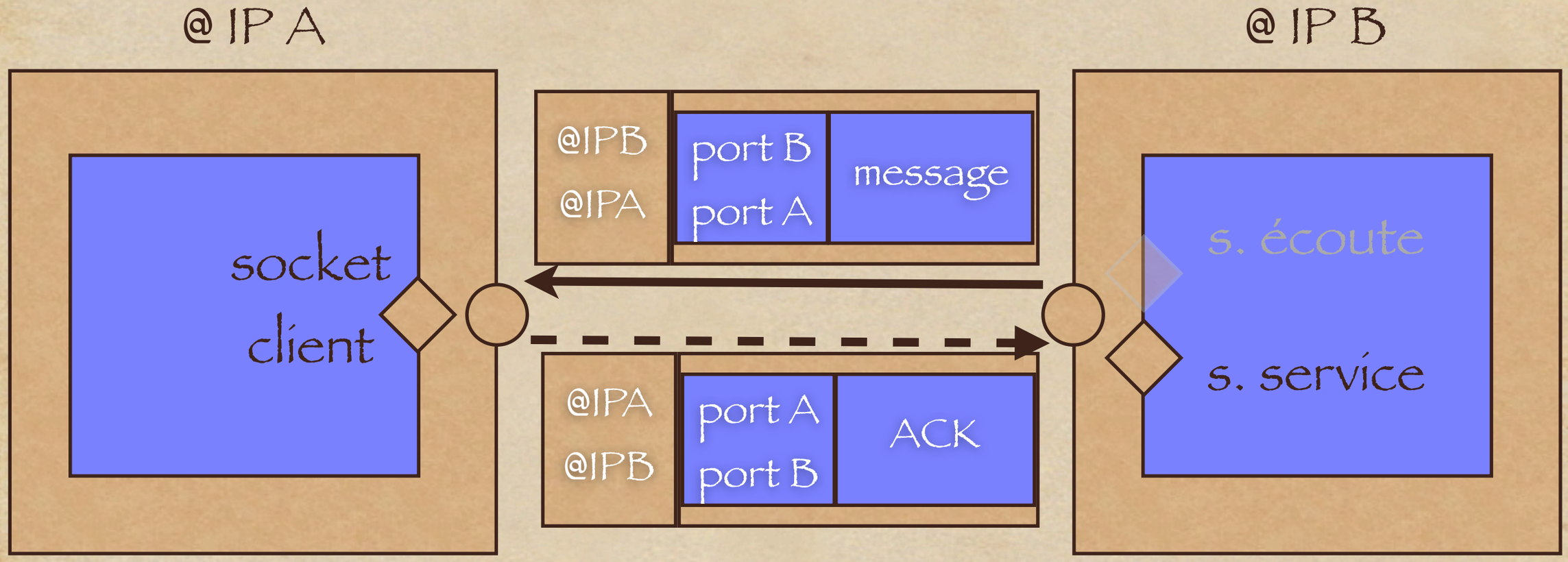
s. écoute

@ loc = @ IP B ou Any
port loc = port B
@ dist = Any
port dist = 0
proto = TCP

s. service

@ loc = @ IP B
port loc = port B
@ dist = @ IP A
port dist = port A
proto = TCP

Communication



socket client

@ loc = @ IP A
port loc = port A
@ dist = @ IP B
port dist = port B
proto = TCP

s. écoute

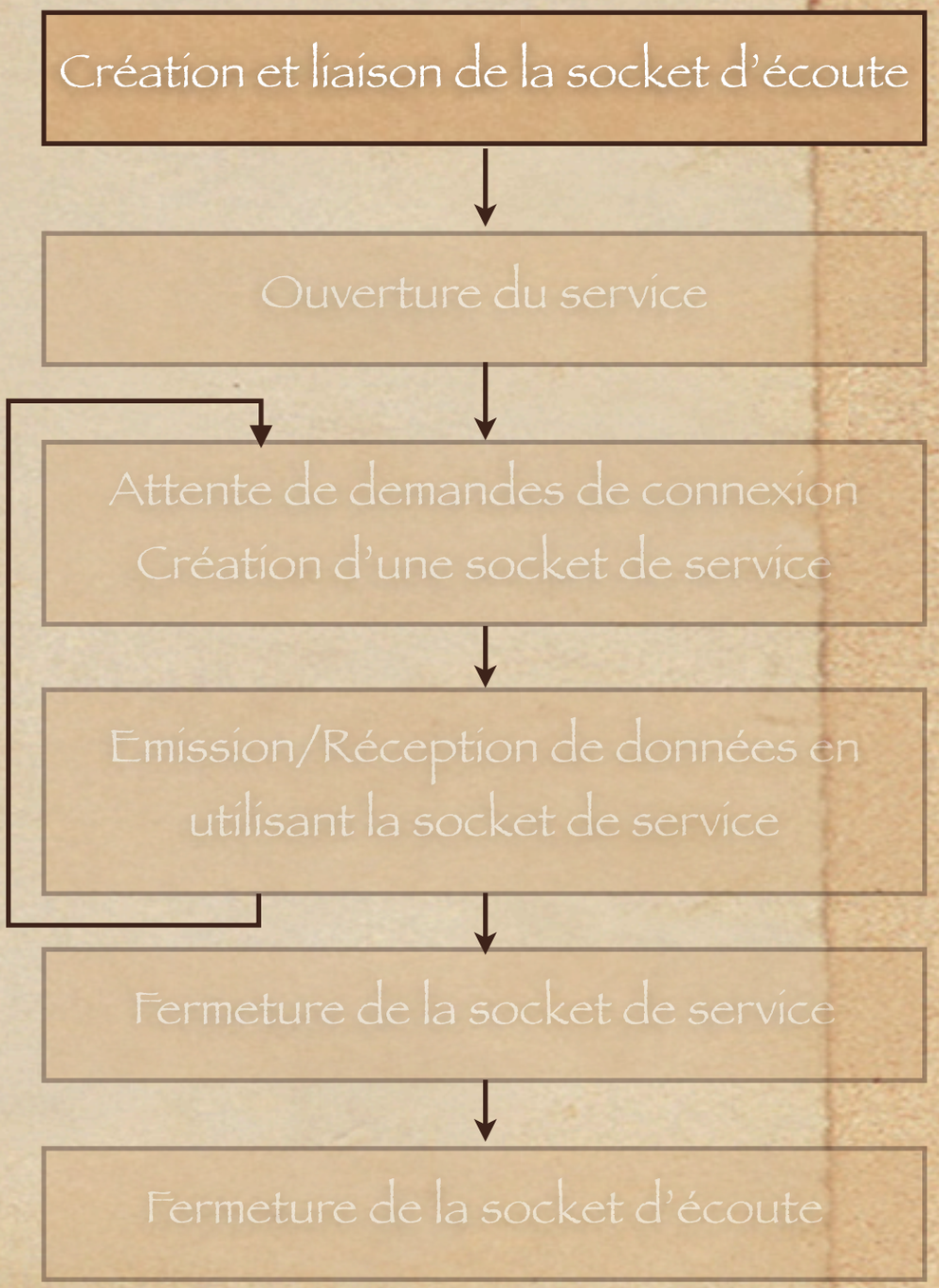
@ loc = @ IP B ou Any
port loc = port B
@ dist = Any
port dist = 0
proto = TCP

s. service

@ loc = @ IP B
port loc = port B
@ dist = @ IP A
port dist = port A
proto = TCP

Prog. en C/C++ : TCP Serv

- Fonction getaddrinfo
  - Gère les adresses IP et les numéros de port
- Fonction socket
  - Crée une socket
- Fonction bind
  - Lie la socket à une adresse IP et à un numéro de port



Prog. en C/C++ : TCP Serv

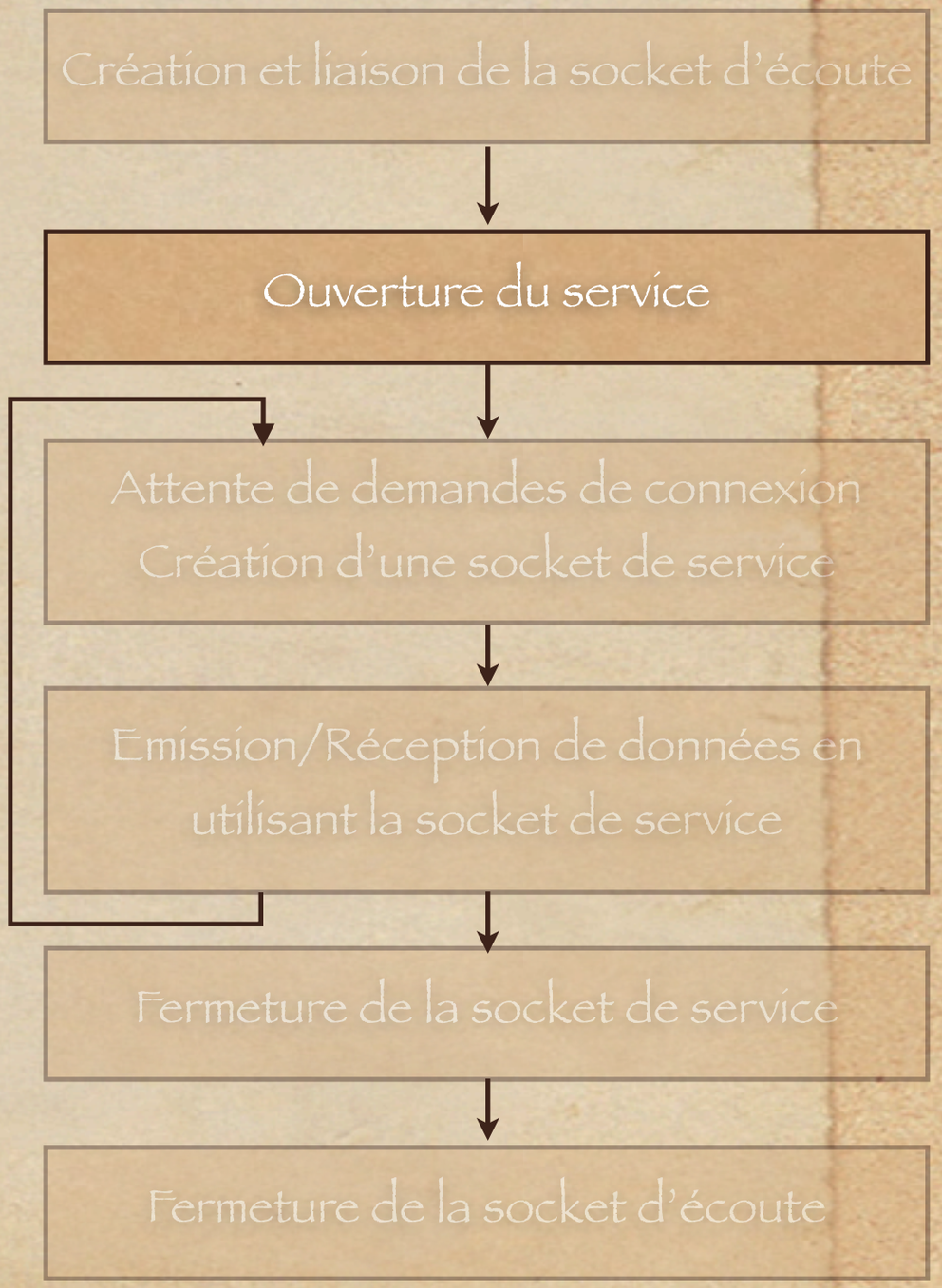
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <memory.h>
#include <sys/socket.h>
#include <netdb.h>

int sock;           //  descripteur de fichier pour la socket
int n; int on = 1;
struct addrinfo     hints, *res, *ressave; // structures gérant les adresses IP v4 et v6
// on initialise la structure à 0 (Windows utilisera memset)
bzero(&hints, sizeof(struct addrinfo));
hints.ai_flags = AI_PASSIVE; // on veut faire un serveur (i.e. ouverture passive)
hints.ai_family = AF_UNSPEC; // on laisse le système choisir IPv4 ou IPv6
hints.ai_socktype = SOCK_STREAM; // on veut utiliser TCP
// on est veut utiliser l'adresse IP Any (0.0.0.0 ou 0::0) et le port 13214
if ( (n = getaddrinfo(NULL, "13214", &hints, &res)) != 0)  {
        printf("Initialisation, erreur de getaddrinfo : %s", gai_strerror(n));
        exit(1);
}
ressave = res;
```

Prog. en C/C++ : TCP Serv

```
// le système peut nous renvoyer plusieurs adresses (en général 1 IPv4 et 1 IPv6)
 do {       // on crèe un socket du bon type (Stream et IPv4 ou IPv6)
            sock = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
            if (sock < 0)
                        continue;                       // ça n'a pas marché on essaie la suivante
            // permet de récupérer un port «zombie»
            setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
            // on essaie de lier la socket à l'adresse courante
            if (bind(sock, res->ai_addr, res->ai_addrlen) == 0)
                        break;                          // ça a marché
            close(sock);        // ça n'a pas marché on ferme la socket
} while ( (res = res->ai_next) != NULL); // on passe à l'adresse suivante
// si on atteint la fin, ça ne marche pas.
if (res == NULL) {
            perror("Initialisation, erreur de socket ou bind.");
            exit(1);
}
// conserve la longueur de l'adresse pour accept
int longeurAdr = res->ai_addrlen;
// il faut libérer la mémoire
freeaddrinfo(ressave);
```

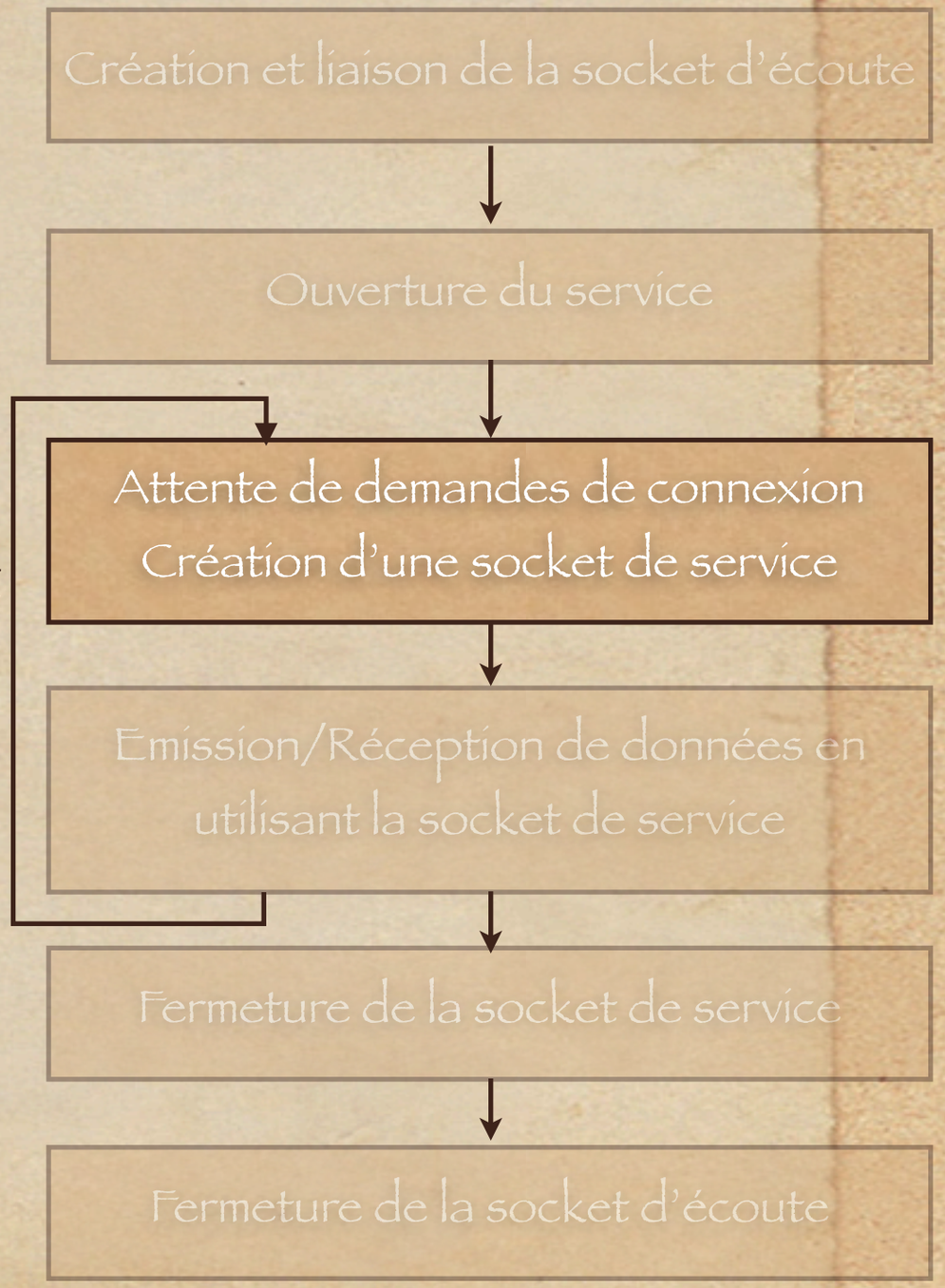
Prog. en C/C++ : TCP Serv

- Fonction listen
  - Commence à écouter des demandes de connexion
  - Pour gérer plusieurs connexions : file d'attente dont on choisit la taille



Prog. en C/C++ : TCP Serv

- Fonction accept
  - Attends une demande de connexion
  - Lors de la réception d'une demande => création d'une socket de service
- Fonction getnameinfo
  - Permet de récupérer une version imprimable de l'adresse IP (ou du nom) et du numéro de port du client

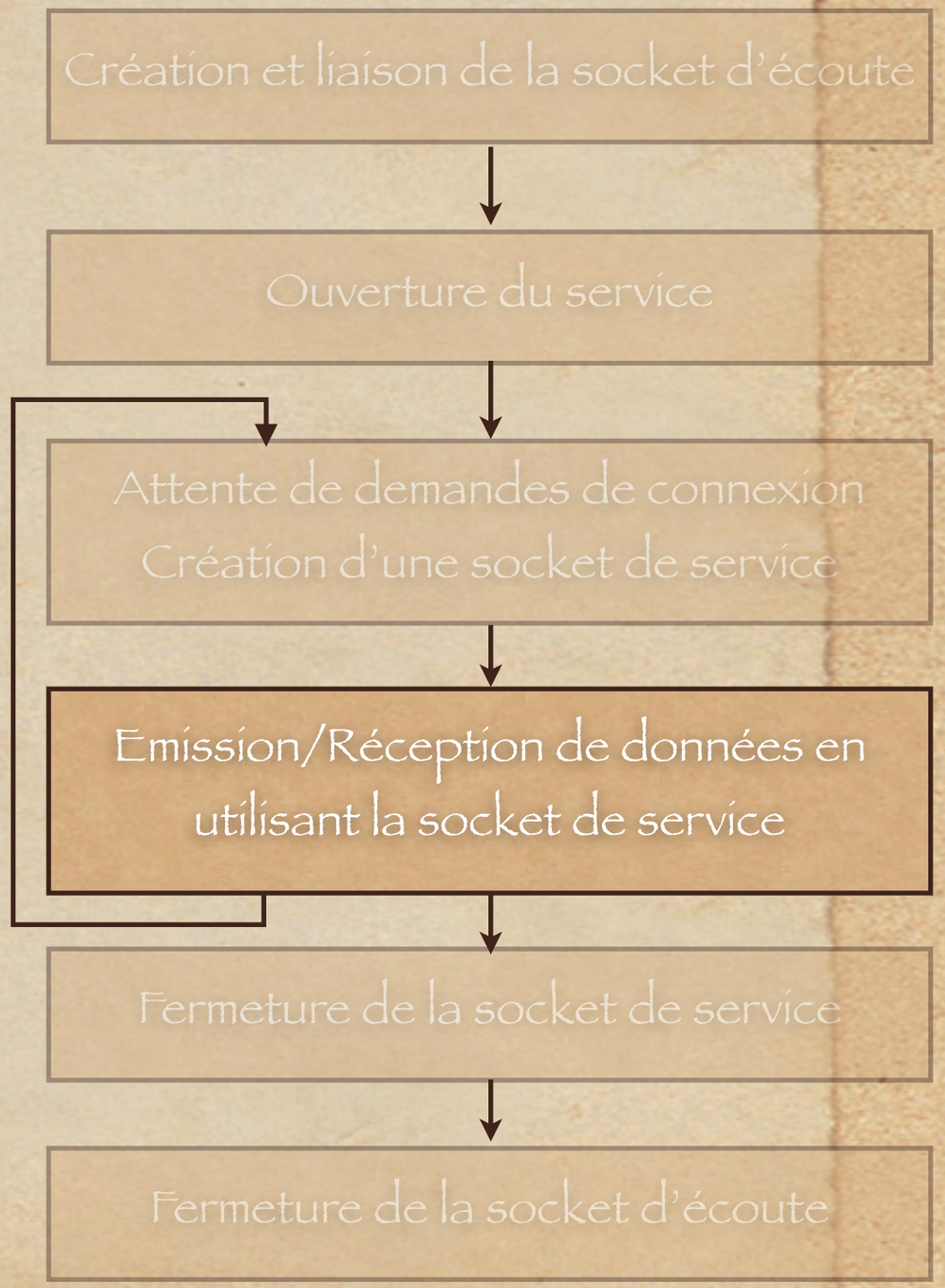


Prog. en C/C++ : TCP Serv

```
int sockEcoute = sock;                  //  La socket créée précédemment servira a attendre
                                        //  et accepter les connexions des clients
int sockService;                        //  Celle ci servira à communiquer avec 1 client
struct sockaddr *clientAddr;
char machine[NI_MAXHOST];
char service[NI_MAXSERV];
// listen avec 5 clients en attente au maximum
listen(socketEcoute, 5); // NB il faudrait vérifier que ça fonctionne
clientAddr = malloc(longeurAdr);
socketService = accept(socketEcoute, clientAddr, &longeurAdr);
if (socketService == -1) {
        perror("Erreur de accept : ");
        exit(1);
}
// on essaie d'avoir le nom de la machine qui s'est connectée
if((n=getnameinfo(clientAddr, longeurAdr, machine, NI_MAXHOST, service,
          NI_MAXSERV, NI_NUMERICHOST|NI_NUMERICSERV)) == 0) {
        printf("Client sur la machine d'adresse %s connecte.\n", machine);
} else {
        printf("Erreur getnameinfo : %s\n", gai_strerror(n));
}
free(clientAddr);
```

Prog. en C/C++ : TCP Serv

- Fonctions : send/recv ou read/write (UNIX seulement)
  - R/W : comme pour tout descripteur de fichier
  - Mais : peuvent être bloquants
  - recv veut dire : on attend puis reçoit une requête client
  - send veut dire : on envoie une réponse au client (bloque si le client n'est pas prêt à recevoir et que sa fenêtre de réception est pleine)



Prog. en C/C++ : TCP Serv

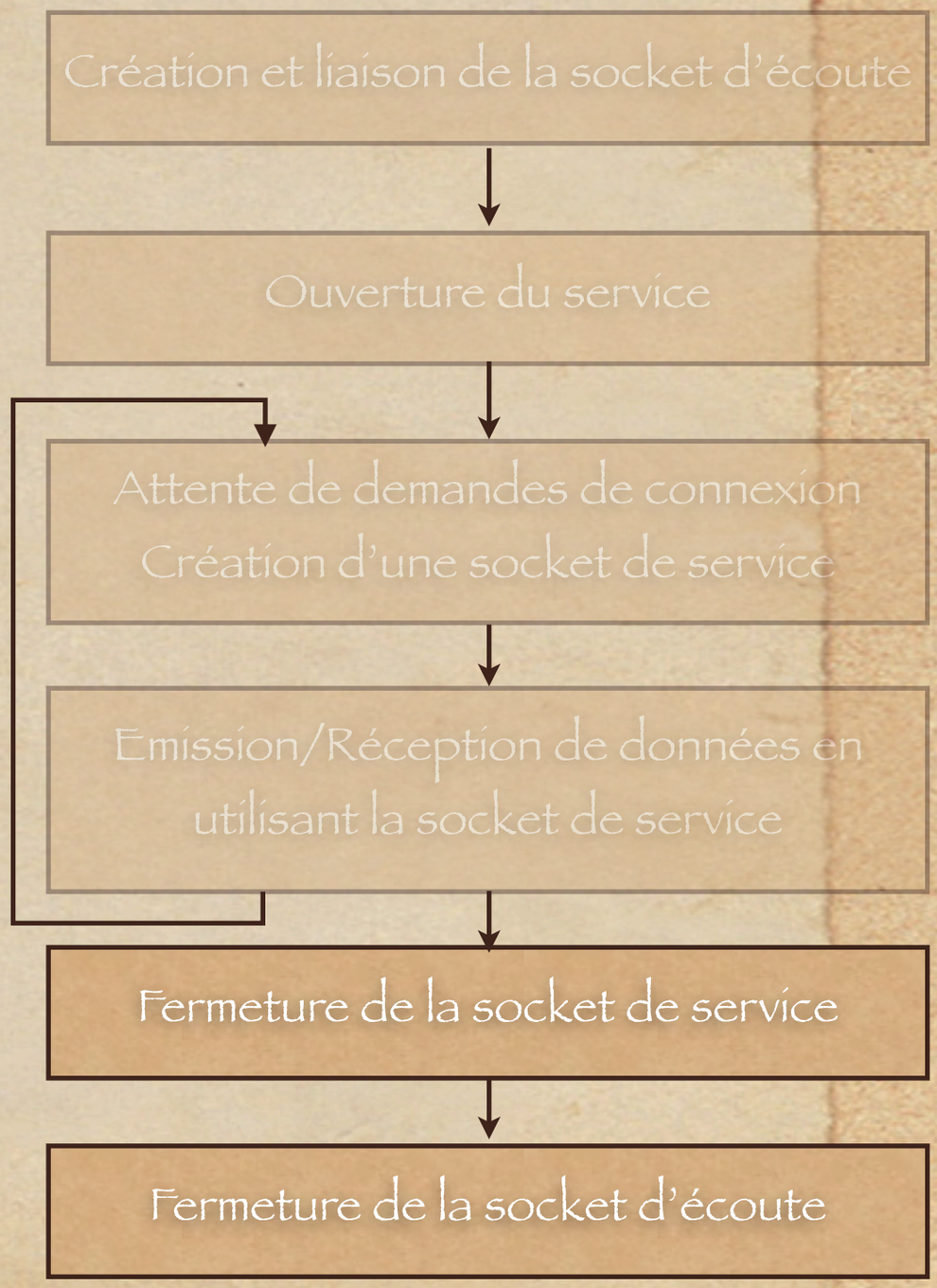
```
int BUFFERLEN = 255;
char buf[BUFFERLEN];          // tampon qui recevra les données reçues
int nbOctetsLus = 0;          // Nombre d'octets lus en tout
int n;                        // Nombre d'octets lus en une fois
// on suppose connu nbOctetsALire (en fonction du protocole)
// note : le 0 final indique qu'on lit des données normalement
// on peut aussi utiliser MSG_OOB pour des données hors bande (1 octet) ou
// MSG_PEEK pour lire des données sans les consommer
while ((n = recv(sockService, buf+nbOctetsLus, BUFFERLEN-nbOctetsLus, 0)) > 0) {
    nbOctetsLus += n;
    if (nbOctetsLus == nbOctetsALire) {
        // on traite le message
         break;
    }
}
if (n < 0) {                          // erreur de recv
    perror("erreur de recv");
    exit(1);
}
if (n == 0) {                         // La connexion a été fermée
    /* ... */
}
```

Prog. en C/C++ : TCP Serv

```
int BUFFERLEN = 255;
char buf[BUFFERLEN];   // tampon d'envoi

sprintf(buf, "requete");    // Ecrit des données dans le tampon
// note : le 0 final indique qu'on envoi des données normalement
// on peut aussi utiliser MSG_OOB pour des données hors bande (1 octet) ou
// MSG_DONTROUTE pour éviter d'utiliser la table de routage
// (donc on est sûr qu'on envoie les données à une machine de notre LAN)
if (send(sockService, buf, 1+strlen(buf),0) == -1) {     // Envoi des données
        perror("erreur de send");
        exit(1);
}
```

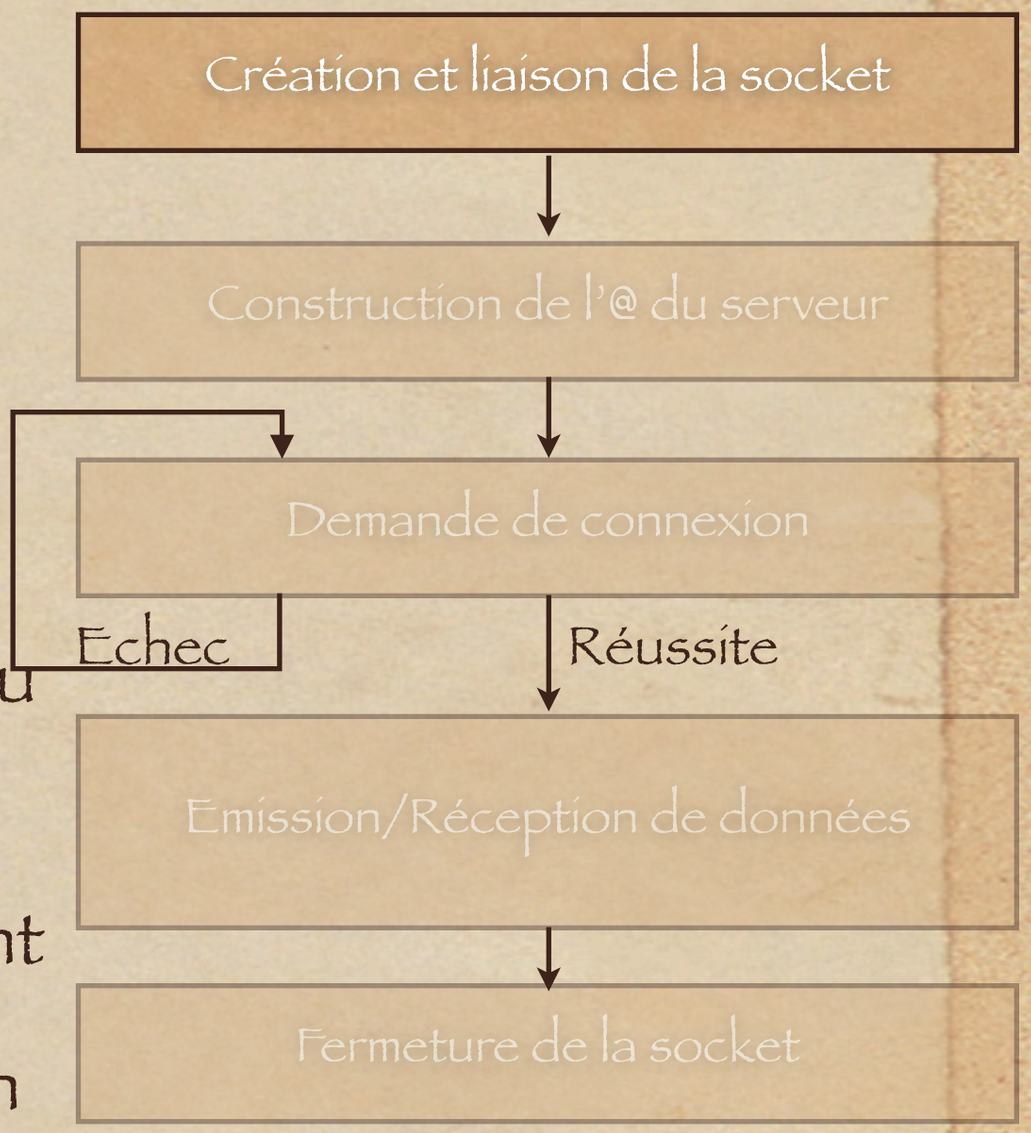
Prog. en C/C++ : TCP Serv

- Fonction : close
  - Comme pour tout descripteur de fichier
  - code : close(sock)



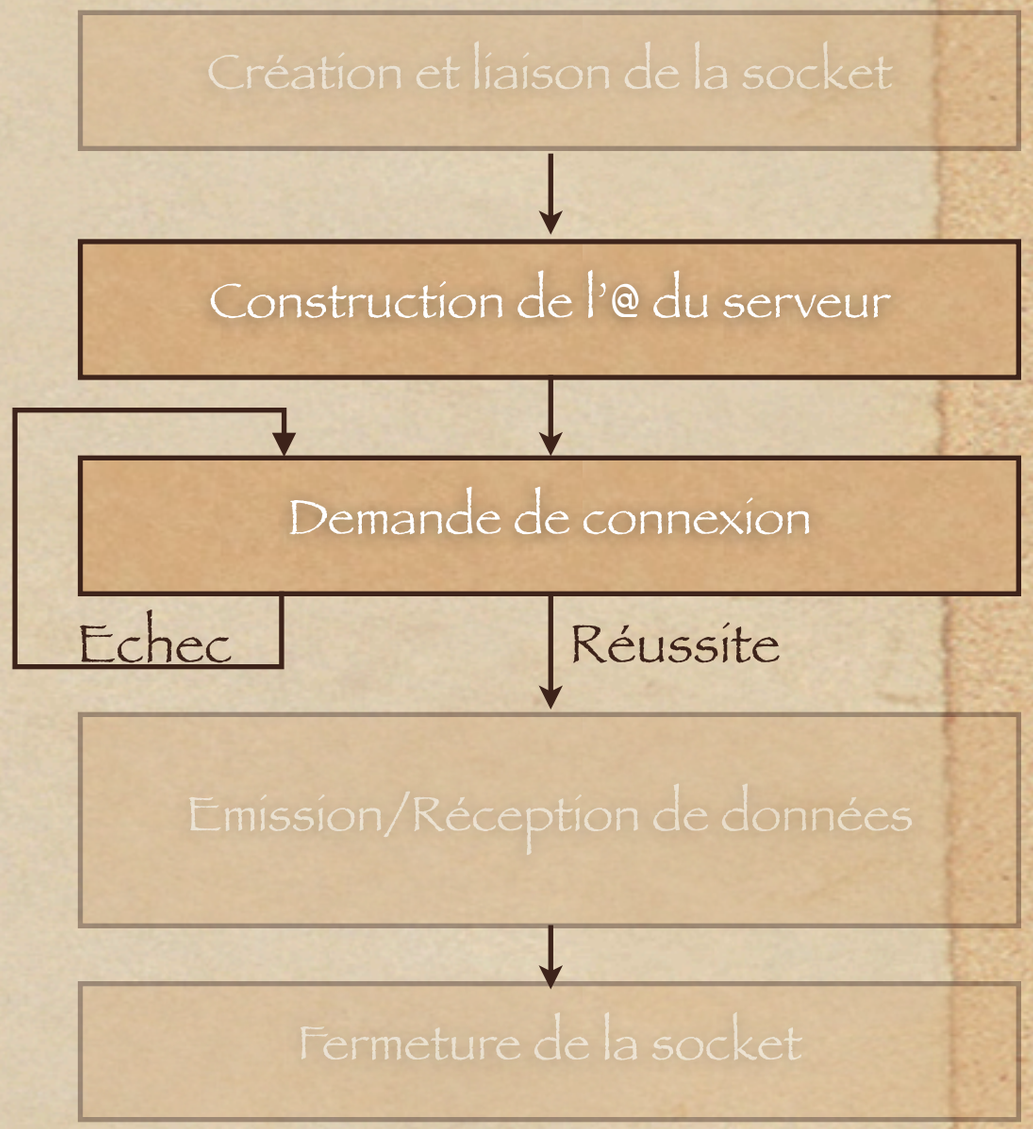
Prog. en C/C++ : TCP Client

- Fonction : socket
  - Idem serveur
- Fonction : bind
  - Optionnel
  - Permet de choisir l'interface réseau a utiliser pour sortir
  - Le port est choisi automatiquement
  - Préciser 0 comme port si on fait un bind



Prog. en C/C++ : TCP Client

- Fonction getaddrinfo
  - Gère les adresses IP et les numéros de port
- Fonction connect
  - Envoie la demande de connexion



Prog. en C/C++ : TCP Client

```
int socketClient; // le socket client
 int n;
struct addrinfo         hints, *res, *ressave;
 // init à 0
bzero(&hints, sizeof(struct addrinfo));
hints.ai_family = AF_UNSPEC; // le système choisit IPv4 ou v6
hints.ai_socktype = SOCK_STREAM; // On veut TCP
 // Note : ici ce sera pour une ouverture active (i.e. avec connect)
 // machine peut contenir une IP ("10.1.0.1" ou "2001:cdba::3257:9652")
 // ou un nom ("image1.edu.ups-tlse.fr")
 // à la place du port (ici "13214" on peut demander un service : "http")
if ( (n = getaddrinfo(machine, "13214", &hints, &res)) != 0)  {
          printf("Initialisation, erreur de getaddrinfo : %s", gai_strerror(n));
          exit(1);
}
ressave = res;
```

Prog. en C/C++ : TCP Client

```
// on teste les différentes adresses
do {      // on crée le socket
          socketClient = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
          if (socketClient < 0)
                    continue;       // ça n'a pas marché, on passe au suivant
          // on essaie de se connecter
          if (connect(socketClient, res->ai_addr, res->ai_addrlen) == 0)
                    break;          // ça a marché
          close(socketClient);   // on ferme le socket qui n'a pas marché
} while ( (res = res->ai_next) != NULL); // on passe à l'adresse suivante
  // si on arrive à la fin c'est qu'aucune adresse ne marche
if (res == NULL) {
          perror("Initialisation, erreur de connect.");
          exit(1);
}
  // on libère la mémoire
freeaddrinfo(ressave);
```

Prog. en C/C++ : TCP

- Remarques
  - Le serveur ne traite qu'un client à la fois (les autres attendent)
- Solutions :
  - Utiliser plusieurs processus (fork) ou des threads (pthread_create)
  - Utiliser des entrées/sorties non bloquantes (pas très efficace : on teste chaque fd régulièrement)
  - Mutliplexer les entrées/sorties avec select

Prog. en C/C++ : TCP

- Select
  - Utilisation : on bloque en attente de données sur plusieurs FD et lorsqu'un est prêt on utilise recvfrom (ou read)
  - int select(int maxfd, fd_set *readset, fd_set *writeset, fd_set *exceptset, const struct timeval *timeout);
  - maxfd : permet de préciser le nombre maximal de fd à tester (exemple pour les fds : 1, 4 et 5 on écrira 6 car il faudra tester de 0 à 5)
  - fd_set : ensembles de descripteurs de fichiers en lecture (readset), écriture (writeset) et exception (exceptset - correspond aux données MSG_OOB)
  - timeout : permet de spécifier une durée maximum d'attente

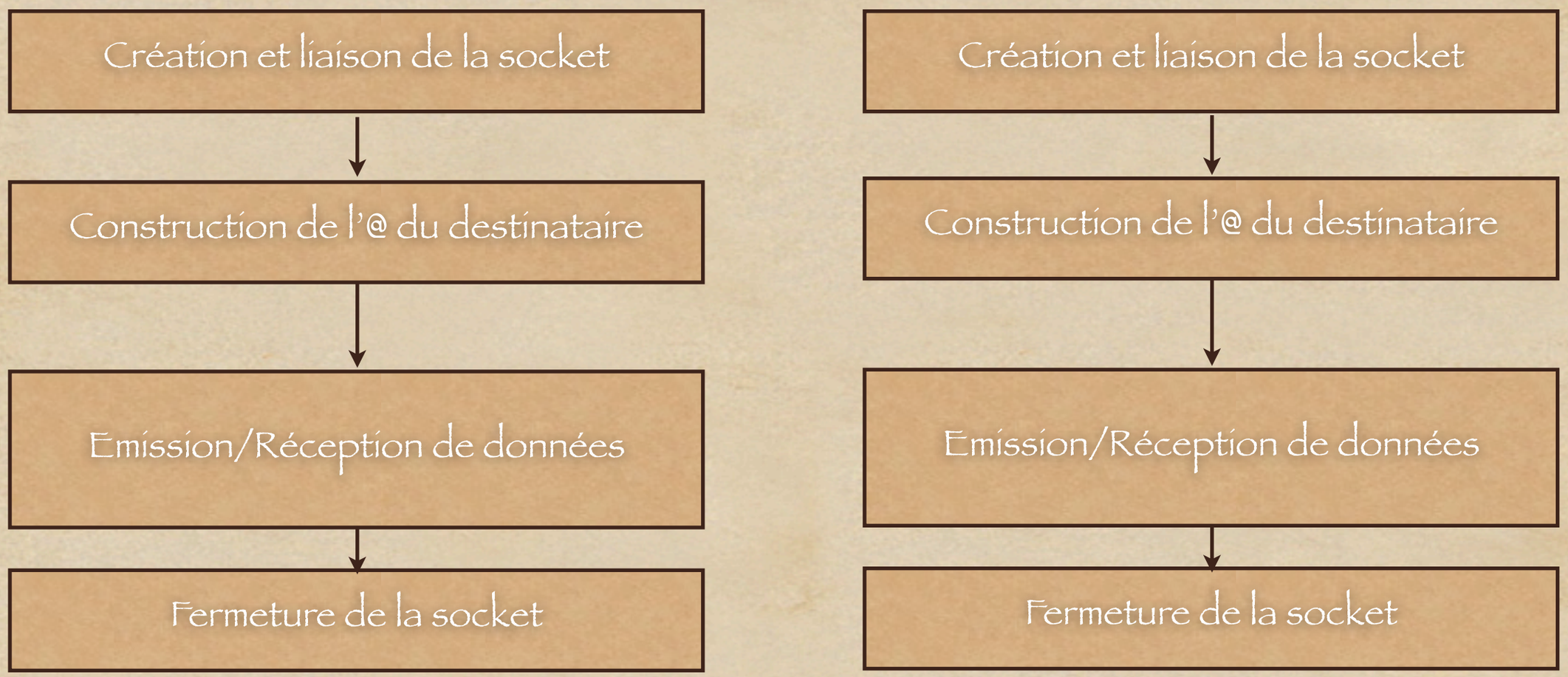
Prog. en C/C++ : TCP

- Select
- Pour manipuler les fd_set on utilise :
  - FD_ZERO(&set) : pour ré-initialiser
  - FD_SET(fd, &set) : pour ajouter le descripteur fd à l'ensemble
  - FD_CLR(fd, &set) : pour enlever fd de l'ensemble
  - FD_ISSET(fd, &set) : renvoie 1 si le fd est positionné
- A la fin de l'attente, select retourne le nombre de descripteurs prêts.
- Il faut ensuite utiliser FD_ISSET pour trouver le/les descripteur(s) prêt(s).
- Note : le timeout n'est pas mis à jour par select. Si on veut savoir combien de temps il reste, il faut le gérer "à la main"

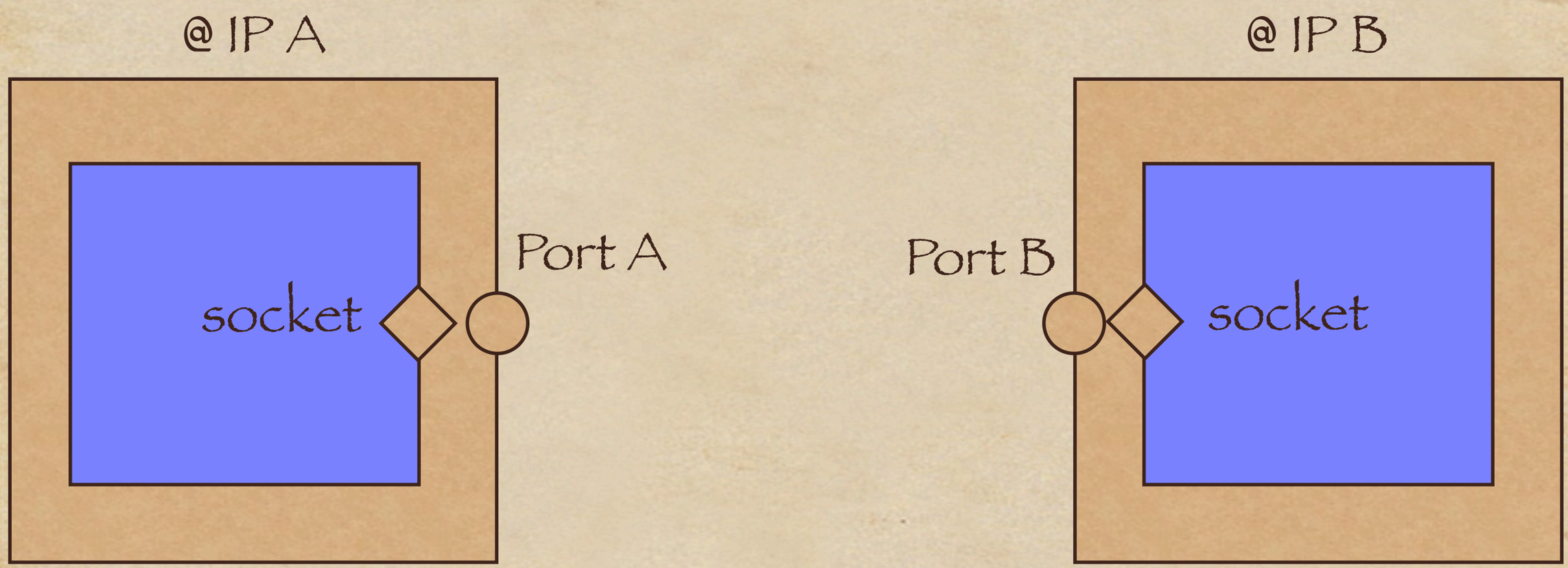
Prog. en C/C++ : TCP

```
// Exemple avec un seul socket :
fd_set fdset;
struct timeval date;
// attendre au max pendant
// 50 ms = 50000 us
date.tv_sec = 0;
date.tv_usec = 50000;

FD_ZERO(&fdset);
FD_SET(sock, &fdset);

// un message est-il arrivé ?
if (select(sock+1, &fdset, NULL, NULL,
                    &date) > 0) {
    int cnt;
    char message[1024];

    // on recupere le message
    cnt = recv(sock, message, sizeof(message),
                                0);
    if (cnt < 0) {
        perror("erreur de recv");
        exit(1);
    }
    // on traite le message ici
     ...
}
```

Emission/réception avec UDP



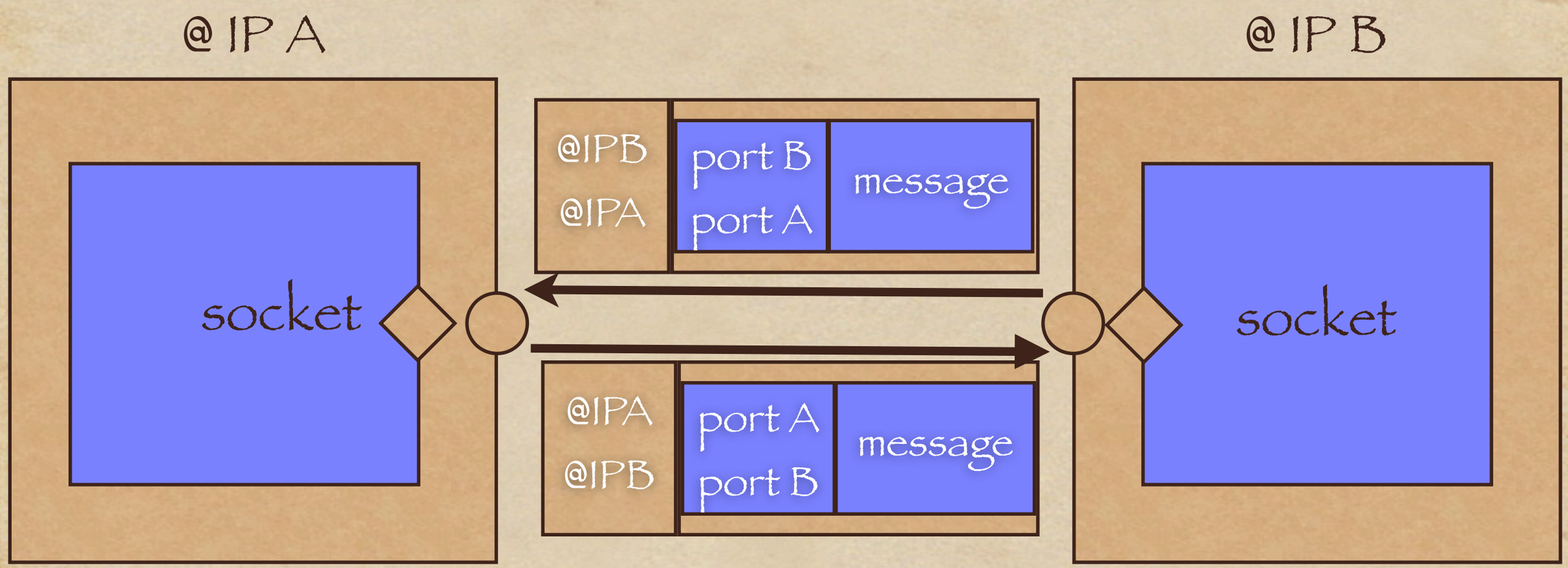
Création sockets + bind



@ loc = @ IP A ou Any
port loc = port A
@ dist = Any
port dist = 0
proto = UDP

@ loc = @ IP B ou Any
port loc = port B
@ dist = Any
port dist = 0
proto = UDP

Communication



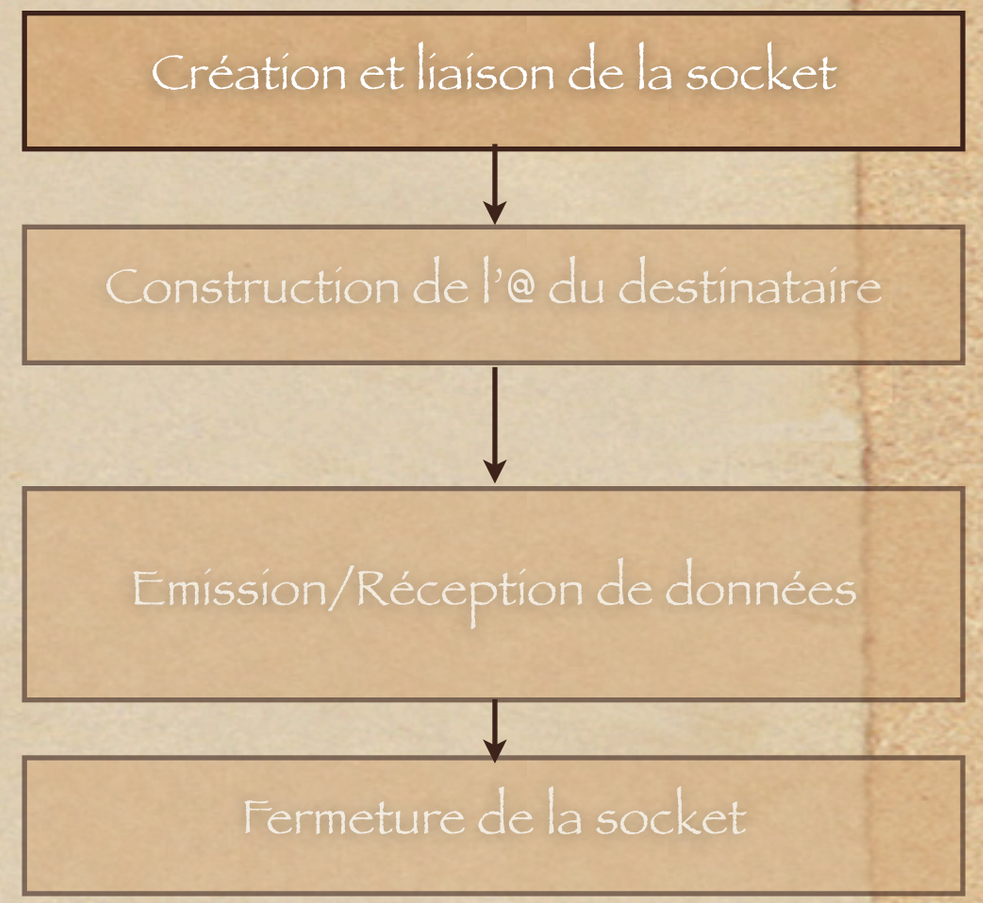
@ loc = @ IP A ou Any
port loc = port A
@ dist = Any
port dist = 0
proto = UDP

Pas de connexion,
en général on précise
la destination à chaque envoi

@ loc = @ IP B ou Any
port loc = port B
@ dist = Any
port dist = 0
proto = UDP

Prog. en C/C++ : UDP

- Fonction getaddrinfo
  - Gère les adresses IP et les numéros de port
- Fonction socket
  - Crée une socket
- Fonction bind
  - Lie la socket à une adresse IP et à un numéro de port



Prog. en C/C++ : UDP

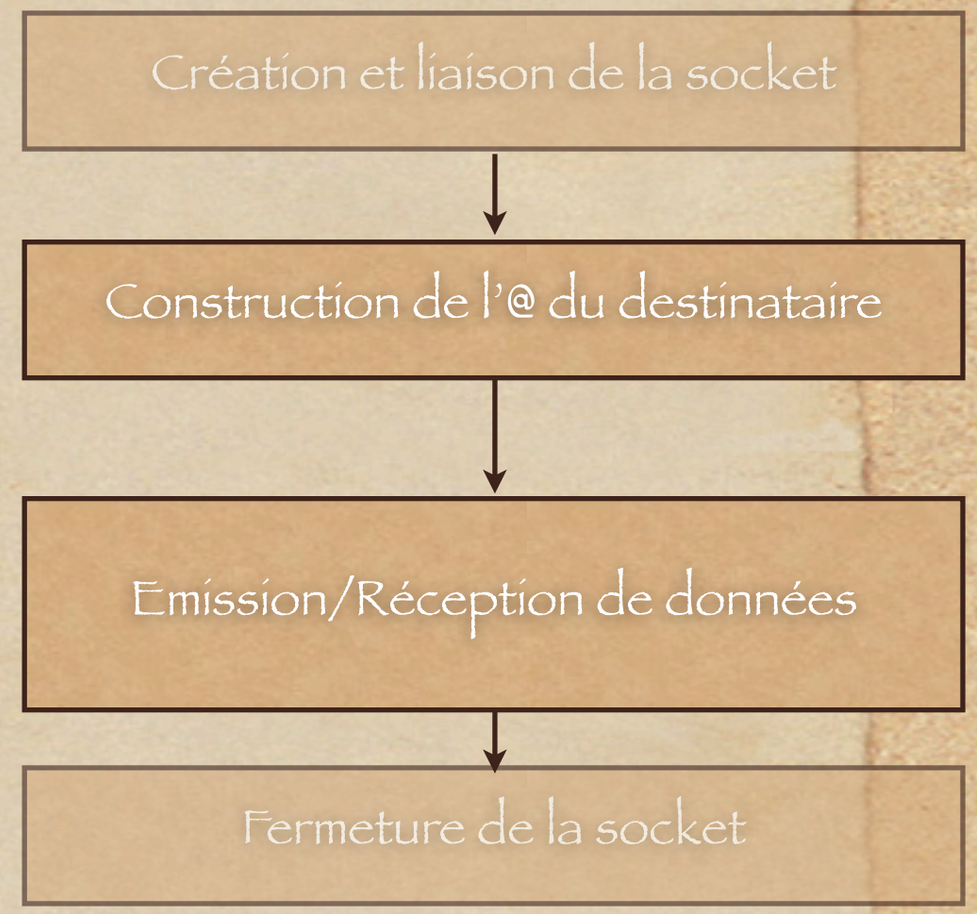
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <memory.h>
#include <sys/socket.h>
#include <netdb.h>

int sock;           //  descripteur de fichier pour la socket
struct addrinfo     hints, *res, *ressave;
socklen_t longueurAdr; int n; short familleAdr;
// init à 0
bzero(&hints, sizeof(struct addrinfo));
hints.ai_family = AF_UNSPEC; // Le système choisit IPv4 ou IPv6
hints.ai_socktype = SOCK_DGRAM; // On veut UDP
// NULL indique qu'on veut Any (0.0.0.0 ou 0::0)
if ( (n = getaddrinfo(NULL, "13214", &hints, &res)) != 0) {
        printf("erreur getaddrinfo : %s\n", gai_strerror(n));
        exit(1);
}
ressave = res;
```

Prog. en C/C++ : UDP

```
do {    // on essaie de créer une socket de ce type
        sock = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
        if (sock < 0)
                continue;                       // erreur, on passe à l'adresse suivante
        // note : si on ne veut qu'émettre il est inutile de faire le bind
        // on essaie de lier la socket à l'adresse courante
        if (bind(sock, res->ai_addr, res->ai_addrlen) == 0)
                break;                          // ça a marché
                close(sock);                    // erreur de bind, on ferme la socket
} while ( (res = res->ai_next) != NULL);
// si aucune adresse n'a marché
if (res == NULL) {
        perror("erreur bind ou socket : ");
        exit(1);
}
// on garde la longueur de l'adresse pour les recvfrom
longueurAdr = res->ai_addrlen;
// on garde la famille pour les sendto
familleAdr = res->ai_family;
printf("Longueur adr : %d\n", longueurAdr); // 28 pour IPv6, 16 pour IPv4
// on libère la mémoire
freeaddrinfo(ressave);
```

Prog. en C/C++ : UDP

- Fonctions : sendto/recvfrom
  - recvfrom peut être bloquant
  - recvfrom veut dire : on attends puis reçois un message (on pourra savoir de qui viens ce message - pour répondre par exemple)
  - sendto permet de préciser l'adresse/port du destinataire du message à envoyer



Prog. en C/C++ : UDP

```
struct sockaddr *adrDest;

bzero(&hints, sizeof(struct addrinfo));
 // on veut une adresse IP de même type que celle du bind/socket
hints.ai_family = familleAdr;
hints.ai_socktype = SOCK_DGRAM;
 // machine peut contenir une IP ("10.1.0.1" ou "2001:cdba::3257:9652")
 // ou un nom ("image1.edu.ups-tlse.fr")
 // à la place du port (ici "13215" on peut demander un service : "http")
if ( (n = getaddrinfo(machine, "13215", &hints, &res)) != 0) {
        printf("erreur getaddrinfo : %s\n", gai_strerror(n));
        exit(1);
}
// on prends la première adresse
 adrDest = (struct sockaddr*)malloc(res->ai_addrlen);
memcpy(adrDest, res->ai_addr, res->ai_addrlen);
// on libère la mémoire
freeaddrinfo(res);
```

Prog. en C/C++ : UDP

```
#define BUFFERLEN 256
char buf[BUFFERLEN];                    // Tampon pour le message
sprintf(buf, "bonjour !");              // On écrit le message dans le tampon

// Envoie le message
if (sendto(sock, buf, strlen(buf) + 1, 0, &adrDest, longeurAdr) <0) {
      perror("erreur de sendto");
      exit(1);
}
```

Prog. en C/C++ : UDP

```
#define BUFFERLEN 256
char buf[BUFFERLEN]; // Tampon pour recevoir le message

struct sockaddr *srcAdr; // Contiendra l'@IP et le port de l'émetteur
srcAdr = (struct sockaddr*) malloc(longueurAdr);
bzero((char *)srcAdr, longueurAdr);  // init. à 0

// Attends et reçoit les données
if (recvfrom(sock, buf, sizeof(buf), 0, srcAdr, &longueurAdr) < 0) {
        perror("erreur de recvfrom");
        exit(1);
}
```

Prog. en C/C++ : UDP

```
char machine[NI_MAXHOST];
char service[NI_MAXSERV];

// on essaie de récupérer l'adresse IP et le port de l'émetteur
if((n = getnameinfo(srcAdr, longueurAdr, machine, NI_MAXHOST,
                    service, NI_MAXSERV,
                    NI_NUMERICHOST|NI_NUMERICSERV)) == 0)
        printf("recu : %s depuis %s:%s\n", buf, machine, service);
else {
        printf("recu : %s\n",buf);
        printf("erreur getnameinfo : %s\n", gai_strerror(n));
}
```

Prog. en C/C++ : diffusion UDP

- Identique à UDP/IP unicast avec deux exceptions :
  - On doit utiliser une adresse de diffusion (broadcast) en IP v4
    - Soit générale : 255.255.255.255
    - Soit spécifique à un LAN : exemple 192.168.1.255 (si on est connecté à plusieurs LANs)

getaddrinfo("255.255.255.255", "13214", &hints, &res)

- Avant de diffuser il faut positionner l'option SO_BROADCAST sur le socket

```
int un = 1;
setsockopt(sock, SOL_SOCKET, SO_BROADCAST, &un, sizeof(un));
```

Prog. en C/C++ : multicast UDP

- Pour envoyer en diffusion restreinte (mcast) :
  - Il faut préciser une adresse multicast comme destination
    - En IP v4 par exemple 225.0.0.1

getaddrinfo("225.0.0.1", "13214", &hints, &res)

    - En IP v6 par exemple ff05::1

getaddrinfo("ff05::1", "13214", &hints, &res)

Prog. en C/C++ : multicast UDP

- On peut modifier la portée de la diffusion
  - En IPv4 : choix du TTL

```
unsigned char ttl = 8;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl));
```

  - En IPv6 : choix du nombre de sauts

```
int hop = 8;
setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_HOPS, &hop, sizeof(hop)));
```

- Par défaut : 1 (limité au réseau local)

Prog. en C/C++ : multicast UDP

- Pour recevoir en diffusion restreinte (mcast) :
  - Il faut s'abonner à une adresse multicast en utilisant setsockopt
    - Pour IPv4 : IP_ADD_MEMBERSHIP
    - Pour IPv6 : IPV6_JOIN_GROUP
- Pour se désabonner on utilisera aussi setsockopt
    - Pour IPv4 : IP_DROP_MEMBERSHIP
    - Pour IPv6 : IPV6_LEAVE_GROUP

Prog. en C/C++ : multicast UDP IPv4

```
bzero(&hints, sizeof(struct addrinfo));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_DGRAM; // on veut UDP
if ( (n = getaddrinfo("225.0.0.1", "13214", &hints, &res)) != 0) {
        printf("erreur getaddrinfo : %s\n", gai_strerror(n)); exit(1);
}

struct ip_mreq mreq;
// on recopie l'adresse IP
memcpy(&mreq.imr_multiaddr, &((struct sockaddr_in *) res->ai_addr)->sin_addr,
                                        sizeof(struct in_addr));
// on le fait pour l'interface par défaut
mreq.imr_interface.s_addr = htonl(INADDR_ANY);

// on rejoint l'adresse IP multicast
if(setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq))!=0) {
        perror("erreur setsockopt"); exit(1);
}
// on libère la mémoire
freeaddrinfo(res);
```

Prog. en C/C++ : multicast UDP IPv6

```
bzero(&hints, sizeof(struct addrinfo));
hints.ai_family = AF_INET6;
hints.ai_socktype = SOCK_DGRAM; // on veut UDP
if ( (n = getaddrinfo("ff05::1", "13214", &hints, &res)) != 0) {
        printf("erreur getaddrinfo : %s\n", gai_strerror(n)); exit(1);
}

struct ipv6_mreq      mreq6;
// on recopie l'adresse IP
memcpy(&mreq6.ipv6mr_multiaddr, &((struct sockaddr_in6 *) res->ai_addr)->sin6_addr,
                sizeof(struct in6_addr));
// on le fait pour l'interface par défaut
mreq6.ipv6mr_interface = 0;

// on rejoint l'adresse IP multicast
if(setsockopt(sock, IPPROTO_IPV6, IPV6_JOIN_GROUP, &mreq6, sizeof(mreq6))!=0) {
        perror("erreur setsockopt"); exit(1);
}
// on libère la mémoire
freeaddrinfo(res);
```

Prog. en C/C++ sous Windows

- Utilisation des Windows Sockets
  - Gèrent en standard : TCP, UDP et IPX (Internetwork Packet Exchange de Novell qui n'est plus très utilisé)
- Les différences avec les Sockets BSD :
  - Il n'y a pas de descripteurs de fichiers sous Windows : on utilise SOCKET au lieu de int comme type.
  - Les fonctions renvoient SOCKET_ERROR au lieu de -1

Prog. en C/C++ sous Windows

- Les différences avec les Sockets BSD (suite) :
  - Avant tout appel à une fonction "socket" il faut initialiser l'API en exécutant la fonction WSAStartup()
  - A la fin il faut (faudrait) utiliser WSACleanup()
  - errno n'est pas disponible (ni perror) : on peut utiliser WSAGetLastError()
  - A la place de bzero on utilise ZeroMemory ou memset (attention à l'inversion des paramètres)

Prog. en C/C++ sous Windows

- Les différences avec les Sockets BSD (suite) :
  - close et ioctl ne sont pas utilisable avec les sockets : à la place on utilise closesocket et ioctlsocket
  - read et write ne peuvent pas s'utiliser pour les sockets : on doit se contenter de send/recv ou sendto/recvfrom

Prog. en C/C++ : UDP Send

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
WSADATA wsaData; // structure pour l'initialisation de WinSock
SOCKET sock;          //  la socket
int  n;
struct addrinfo        hints, *res, *ressave;
int longueurAdr;
struct sockaddr *adrDest;

// initialisation de l'API
if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) { // Winsock 2.2
        fprintf(stderr, "Erreur de WSAStartup() : %d", WSAGetLastError());
        exit(1);
}
```

Prog. en C/C++ : UDP Send

```
// Récupération de l'adresse de destination (@IP+port)
 ZeroMemory( &hints, sizeof(hints) );
hints.ai_family = AF_UNSPEC; // le système choisira IPv4 ou IPv6
hints.ai_socktype = SOCK_DGRAM; // on veut UDP
// à la place de localhost on peut mettre le nom de la machine
// ou son adresse IP (v4 ou v6)
if ( (n = getaddrinfo("localhost", "13214", &hints, &res)) != 0) {
            fprintf(stderr, "erreur getaddrinfo : %s\n", gai_strerror(n));
            exit(1);
}
ressave = res;

do {      // Construction d'une socket compatible avec cette adresse
            sock = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
            if (sock != SOCKET_ERROR)
                        break;              // ça a marché
} while ( (res = res->ai_next) != NULL);
```

Prog. en C/C++ : UDP Send

```
// aucune adresse n'a fonctionné
if (res == NULL) {
        fprintf(stderr, "Erreur de socket() : %d", WSAGetLastError());
        exit(1);
}
// on alloue puis on recopie l'adresse IP + port de destination
adrDest = (struct sockaddr*)malloc(res->ai_addrlen);
memcpy(adrDest, res->ai_addr, res->ai_addrlen);
longueurAdr = res->ai_addrlen;
// on libère la mémoire allouée par getaddrinfo
freeaddrinfo(ressave);

#define BUFFERLEN 256
char buf[BUFFERLEN];       // Tampon pour le message
sprintf_s(buf, BUFFERLEN, "bonjour !");  // On écrit le message dans le tampon

// Envoie le message
if (sendto(sock, buf, strlen(buf) + 1, 0, adrDest, longueurAdr) <0) {
        fprintf(stderr, "Erreur de sendto() : %d", WSAGetLastError());
        exit(1);
}

closesocket(sock);
WSACleanup();
```