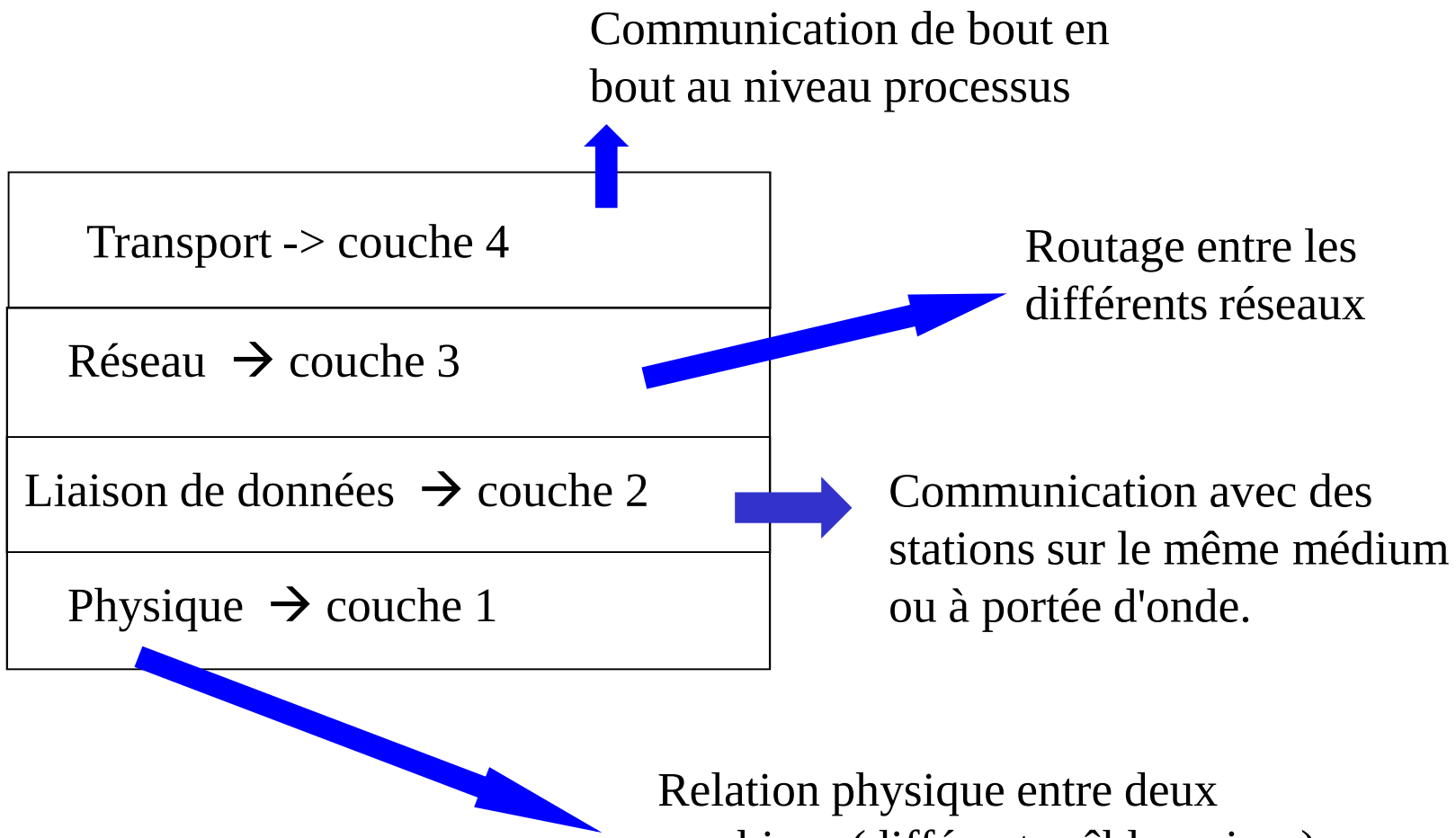
Les Sockets

- Généralités
- Programmation en IPv4
  - En langage C
  - En Java
- Programmation en IPv6

Quelques rappels



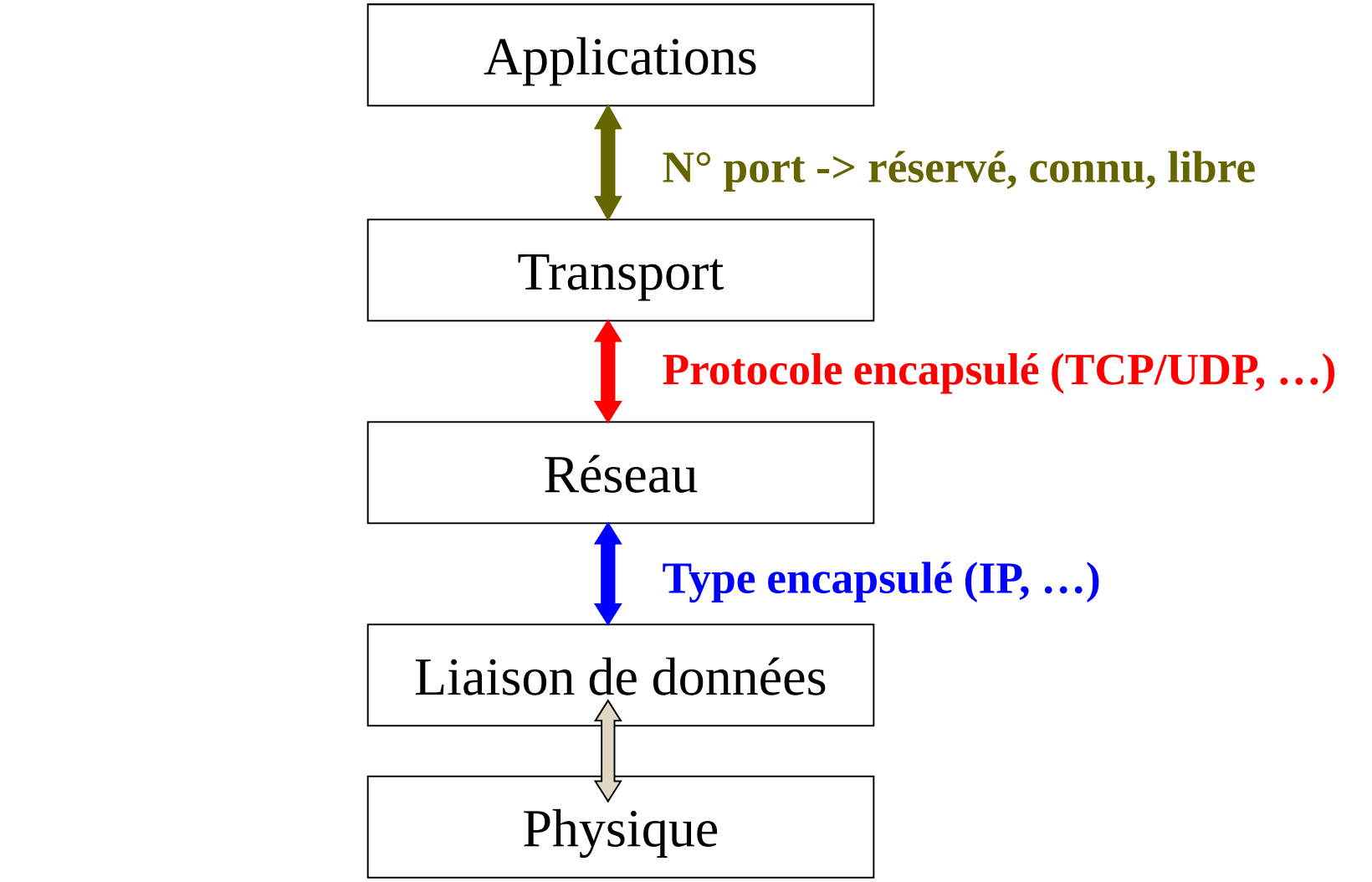
Généralités sur les sockets(1)

- Mécanisme d'interface de programmation
  - permet aux processus d'échanger des données
  - n'implique par forcément une communication par le réseau (ex socket unix)

- Une connexion est entièrement définie sur chaque machine par :
  - le type de protocole (TCP, UDP,…)
  - l'adresse IP
  - le numéro de port associé au processus

    → (statiquement pour le serveur, dynamiquement pour le client)

Généralités (2)



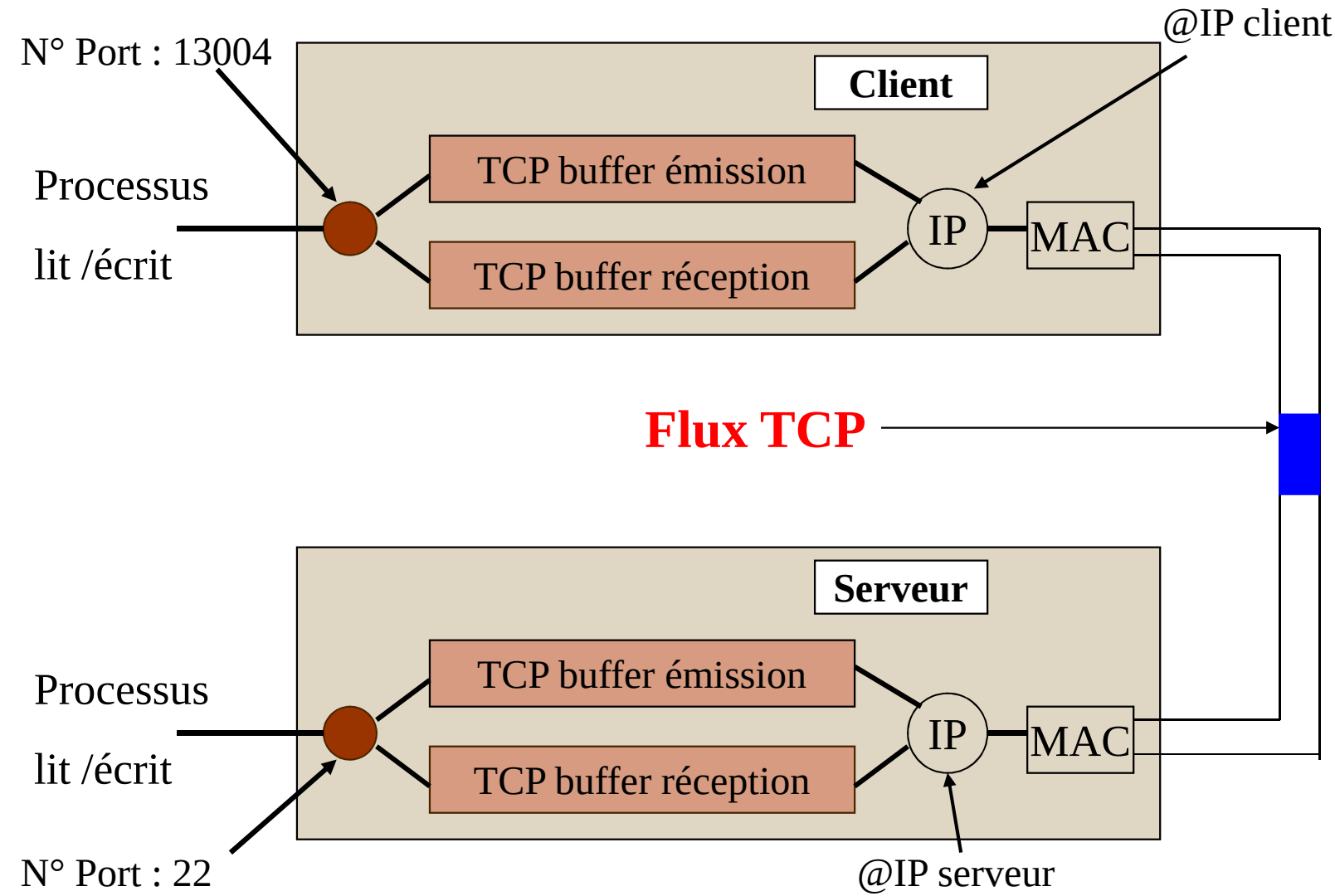
Mode connecté/ non connecté

- Mode connecté (TCP)
  - Problèmes de communication gérés automatiquement
  - Gestion de la connexion coûteuse en message
  - Primitives simples d'émission et de réception
  - Pas de délimitation des messages dans le tampon

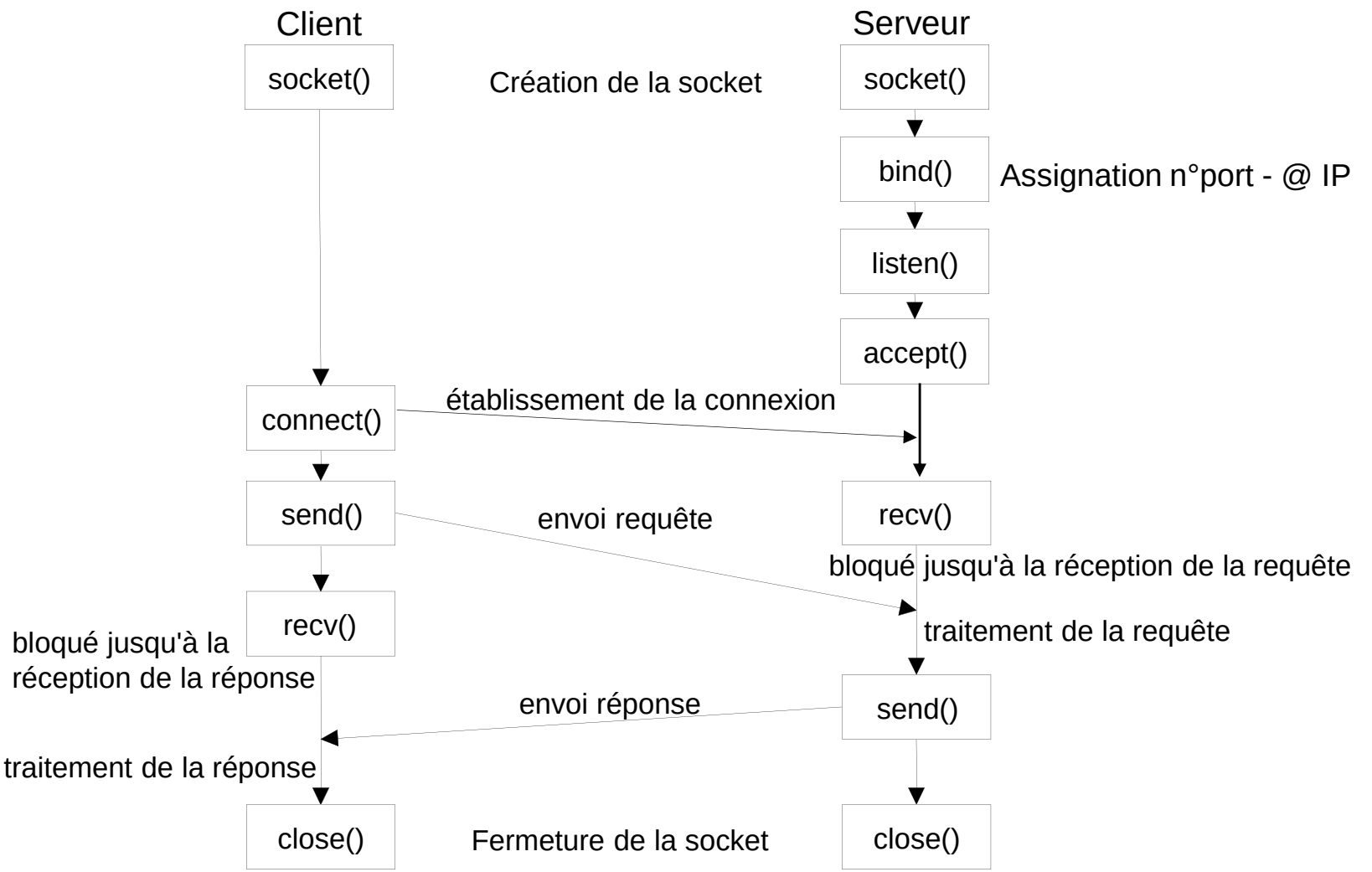
- Mode non connecté (UDP)
  - Consomme moins de ressources
  - Permet la diffusion
  - Aucune gestion des erreurs,

    ➡ c'est la couche applicative qui doit gérer ce problème

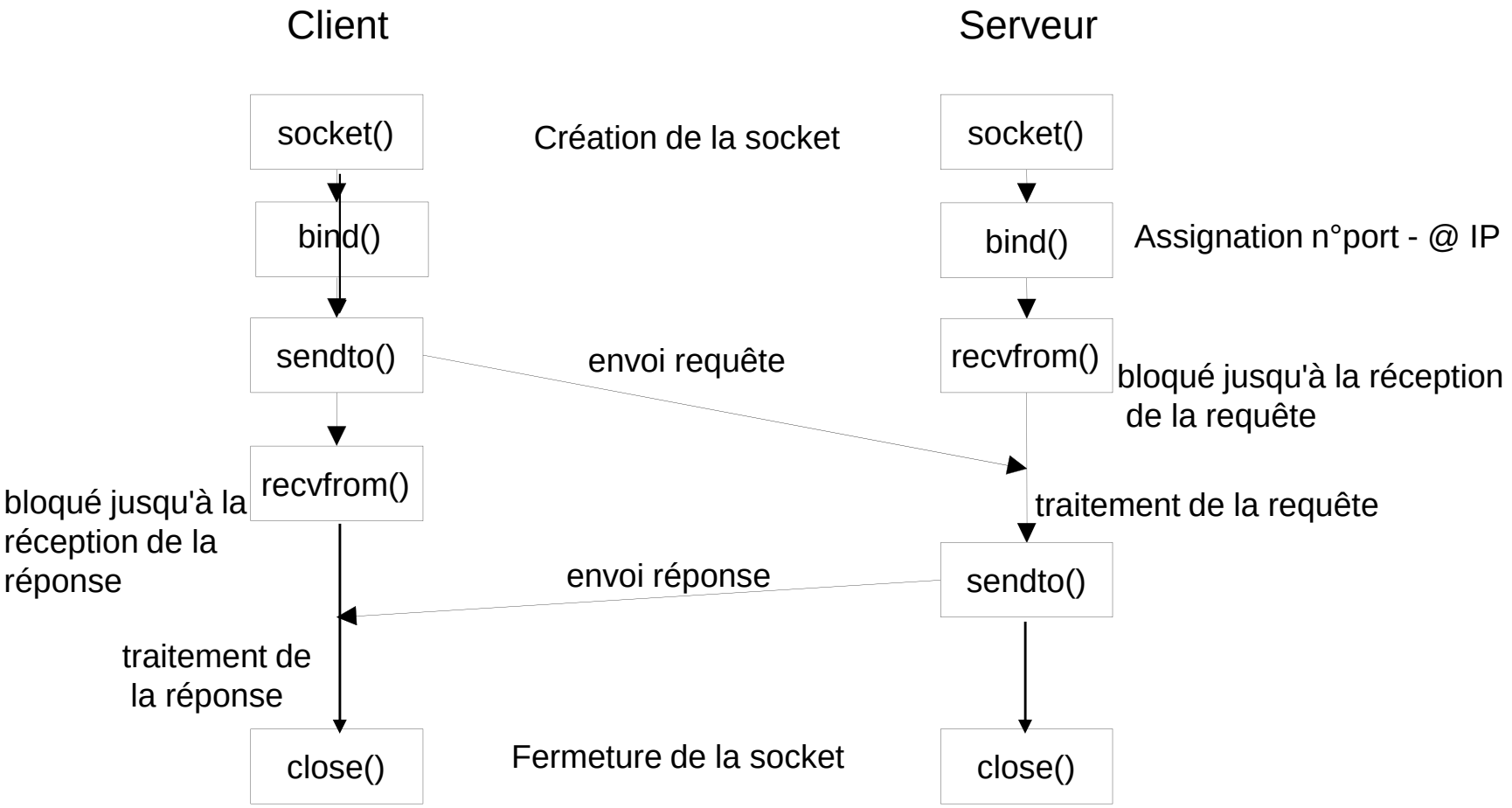
Une connexion TCP



Mode Connecté



Mode non connecté



Généralités (3)

- Une connexion
  - @IP source, @IP destination, port source, port destination
- Une socket est un fichier virtuel sous Unix avec les opérations d'ouverture, fermeture, écriture, lecture …

  (concept légèrement différents sous windows)
- Ces opérations sont des appels systèmes
- Il existe différents types de sockets:
  - Stream socket (connection oriented) ➔ SOCK_STREAM

    mode connecté = TCP
  - Datagram sockets (connectionless) ➔ SOCK_DGRAM

    mode non connecté = UDP
  - Raw sockets ➔ SOCK_RAW

    accès direct au réseau (IP, ICMP)

Programmation en C (1)

- Définition d'une socket
  - **int s = socket (domaine, type, protocole)**
    - *Domaine*
      - AF_UNIX : communication interne à la machine
        structure sockaddr_un dans <sys/un.h>
      - ***AF_INET : communication sur internet en utilisant IP***
    - *Type*
      - ***SOCK_STREAM : mode connecté (TCP)***
      - ***SOCK_DGRAM : mode non connecté (UDP)***
      - SOCK_RAW : utilisation directe niveau 3 (IP, ICMP,…)
    - *Protocole*
      - Actuellement non utilisé, valeur = 0.

Programmation en C (2)

- Attachement d'une socket
  - **int error = bind (int sock, struct sockaddr *p, int lg)**
    - *error :* entier qui contient le compte-rendu de l'instruction
      - 0 : opération correctement déroulée
      - -1 : une erreur est survenue (en général, problème dans p)
    - *sock :* descripteur de la socket
    - *p :* pointeur vers la zone contenant l'adresse de la station
    - *lg :* longueur de la zone p

Fonction définit avec sockaddr,
→ *utilisation réelle de sockaddr_in (AF_INET) ou sockaddr_un (AF_UNIX)*

Programmation en C (3)

- Adressage (inclure fichier <sys/socket.h> et <netinet/in.h>)
  - **struct sockaddr_in {**
    short sin_family; ← domaine
    u_short sin_port; ← n° port
    struct in_addr sin_addr; } ← @IP système

  - *sin_port* = numéro de port
    - soit laissé libre
    - soit affecté : htons (n°port) (conversion short -> réseau)

  - *Struct in_addr sin_addr*
    - Si serveur sin_addr.s_addr = INADDR_ANY
    - Si client

```
struct hostent *hp;
hp = gethostbyname(" …. ");
bcopy(hp->h_addr, &x.sin_addr, hp->h_length);
```

Programmation en C (4)

Primitives du serveur

- **int error = listen (int sock, int taille)**
  - *error :* entier qui contient le compte-rendu de l'instruction
    - 0 : opération correctement déroulée , -1 erreur
  - *taille* : nombre de requêtes maximum autorisées

➡ Permet de créer une file d'attente pour recevoir les demandes de connexion qui n'ont pas encore été prises en compte

- **int scom = accept (int sock, struct sockaddr *p, int *lg)**
  - *struct sockaddr :* stockage de l'adresse appelant
  - *lg* : longueur de l'adresse appelant
  - *scom* : nouvelle socket permettant la communication

➡ **Primitive bloquante**

Programmation en C (5)

- Ouverture d'une connexion, côté client
  - **int error = connect (int sock, struct sockaddr *p, int lg)**
    - *error* : 0 : opération correctement déroulée , -1 erreur
    - *struct sockaddr* : adresse et port de la machine distante (serveur)
    - *lg* : taille de l'adresse de la machine distante

- Fermeture d'une connexion
  - **int close (int sock)**

    → le plus utilisé
  - **int shutdown (int sock, int how)**
    - *how* : 0 : réception désactivé,
      1 : émission désactivé,
      2 : socket désactivé

Programmation en C (6)

- Emission de données en mode connecté
  - **int send (int sock, char *buf, int lg, int option)**
    - *option :* 0 rien, MSG_OOB pour les messages urgents
  - **int write (int sock, char *buf, int lg)**
    - utilisable seulement en mode connecté, permet d'écrire dans un descripteur de fichier
- Emission de données en mode non connecté
  - **int sendto (int sock, char *buf, int lg, 0, struct sockaddr *p, int lg_addr)**
    - *p :* contient l'adresse du destinataire
    - *lg_addr :* longueur de l'adresse destinataire

Programmation en C (7)

- Réception de données en mode connecté
  - **int recv (int sock, char *buf, int lg, int option)**
    - *option :* 0 rien, MSG_OOB pour les messages urgents, MSG_PEEK lecture des données sans les retirer de la file d'attente
  - **int read (int sock, char *buf, int lg)**
    - utilisable seulement en mode connecté, renvoie le nombre d'octets réellement lus.
- Reception de données en mode non connecté
  - **int recvfrom (int sock, char *buf, int lg, 0, struct sockaddr *p, int lg_addr)**
    - *p :* contient l'adresse de la source
    - *lg_addr :* longueur de l'adresse source

Programmation en C (7)

- Quelques fonctions
  - **int getsockname (int sock, struct sockaddr *p, int lg)**
    - permet de récupérer l'adresse locale d'une socket, et son numéro de port
    - ntohs(p->sin_port) pour afficher le numéro du port
    - inet_ntoa(p->sin_addr.s_addr) pour afficher le nom de la machine

    (Attention : peut provoquer des segmentation fault !!! →getnameinfo())

- Fonctions bloquantes/ non-bloquantes
  - **fcntl(int sock, F_SETFL, O_NONBLOCK)**
  - **ioctl (int sock, FIONBIO, 0/1)** → (0 : bloquant, 1, non bloquant)
    - permet de rendre une socket bloquante ou non bloquante en lecture/écriture
    - si pas de données en lecture, renvoie err=-1 et errno=EWOULDBLOCK

Programmation en C (8)

- Programmation sous windows
  - **Identique sauf obligation d'initialiser la librairie qui gère les sockets :**

```
#include <winsock2.h>
#include <iostream>
#pragma comment(lib, "ws2_32.lib")

WSADATA info;
if (WSAStartup(MAKEWORD(2,0), &info) == SOCKET_ERROR)
{  cout << "erreur dans l'initialisation " <<endl;
            WSACleanup();
            exit(1);
   }
int socket ; struct sockaddr_in adr; struct hostent *entree; …
```

Programmation en java (1)

- Création d'une socket en mode connecté et connexion
  - **Côté client**
    - *Socket sock = new socket (nom_serveur, n°port)*
      - permet la connexion direct en TCP
      - plus de recherche de nom
  - **Côté serveur**
    - *ServerSocket sock = new ServerSocket (n°port)*
      Socket scom = sock.accept()
      - la communication s'établit avec la socket scom
      - Plus besoin d'attachement, c'est automatique

Programmation en java (2)

- Lecture/ écriture sur des sockets en mode connecté

  plusieurs possibilités

  - **cas possible pour une socket soc**
    - *lecture*

      Reader reader = new InputStreamReader(soc.getInputStream())
      BufferedReader texte = new BufferedReader(reader);
      line = texte.readLine();

    - *écriture*

      Printstream sortie = new PrintStream(sock.getOutputStream());
      sortie.println(line);

Programmation en java (3)

- Création d'une socket en mode non connecté
  - **Côté client**
    - *DatagramSocket sock = new DatagramSocket ()*
  - **Côté serveur**
    - *DatagramSocket sock = new DatagramSocket (n°port)*
  - **Emission/réception**
    - *DatagramPacket msg = new DatagramPacket(buffer, taille, serveur, n°port)*
      sock.sent(msg);
    - *DatagramPacket recu = new DatagramPacket(buffer, taille);*
      sock.receive(recu);
      recu.getAddress() -> adresse de l'expéditeur
      recu.getPort() -> N° port de l'expéditeur
      recu.getData() -> les données.

Programmation en java (4)

- Utilisation de la classe InetAdress
  - **Récupération de l'adresse IP de la machine**
    - *InetAddress a = InetAddress.getLocalHost();*
    - *InetAddress a = InetAddress.getByName("….");*

      - getHostName() : nom de la machine
      - getHostAddress() : numéro IP de la machine
      - toString() : affiche les deux informations précédentes.

Programmation IPv6

- Pas de changement pour les langages qui utilisent des couches d'abstraction et qui ne référencient pas les adresses IPv4 directement (Java)

- Pour les autres, les changements sont minimisés….

- Les API restent identiques :
  - socket() utilise AF_INET6
  - bind() connect() accept()
  - send() recv()

Programmation IPv6 en C (1)

- Quelques changements !!
  - **struct sockaddr_in6 {**
    u_char sin6_family; ← domaine
    u_int16m_t sin6_port; ← n° port
    struct in6_addr sin6_addr; ← @IP
    u_int32m_t sin6_flowinfo; … } ← id de flux
  - Pour les conversions, utilisation de *struct sockaddr_storage*
    - Permet le mappage du sockaddr_in
    - Permet le mappage du sockkadr_in6
  - Affectation de l'adresse IP
    - Si serveur, l'adresse d'écoute vaut in6addr_any
      d'où
      memcpy((void *)&adr6.sin6_addr, (void *)&in6addr_any,
      sizeof (in6addr_any));

Programmation IPv6 en C (2)

- Disparition de gethostbyname et de gethostbyaddr
  - **Remplacement par getaddrinfo() et getnameinfo()**

```
int getaddrinfo(
        const char *nodename,           // nom d'une machine
        const char *servname,           // nom du service ou n° port
        const struct addrinfo *hints,   // filtre
        struct addrinfo **res);         // liste de résultat possible


struct addrinfo {
        int ai_flags;                   // AI_PASSIVE, AI_NUMERICHOST,…
        int ai_family;                  // AF_INET….
        int ai_socktype;                // SOCK_...
        int ai_protocol;                // 0 ou IPPROTO_xx pour ipv4 et ipv6
        size_t ai_addrlen;              // taille de l'adresse binaire ai_addr
        char * ai_canonname;            //le fqdn
        struct sockaddr * ai_addr;      // l'adresse binaire
        struct addrinfo *ai_next;       // liste chainée sur la structure
```

Programmation IPv6 en C (3)

getnameinfo -> traduction adresse IP vers nom

```
int getnameinfo(
        const struct sockaddr *sa,          // l'adresse IP obtenu
        socklen_t salen,                    // sa taille
        char *host, size_t hostlen,         // résultat pour le nom
        char *serv, size_t servlen, int flags);      // résultat pour le service
```

- Autres fonctions possibles :
  - inet_ntop et inet_pton qui permettent de convertir une adresse binaire en texte et vice –versa

```
        char * inet_ntop (int af,    // Af_INET ou AF_INET6
                          const void *src,        //adresse binaire
                          char *dst,              //adresse du résultat
                          size_t size)            // taille du tampon
```

  - getifaddrs (struct ifaddrs *) -> permet de récupérer les interfaces réseaux

Plusieurs client possibles

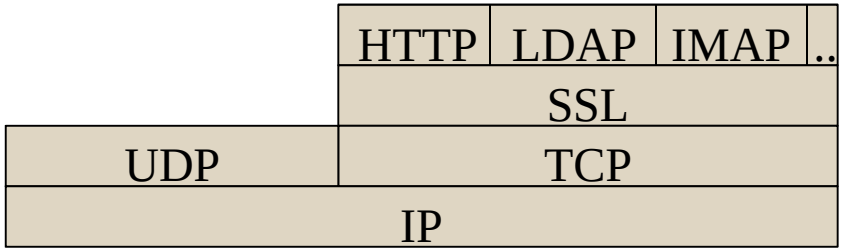
- Actuellement, le serveur ne gère qu'un client à la fois
  → Les autres sont mis en attente
  - Utilisation possible de thread
  - Utilisation de nouveaux processus (fork)
    (après le accept)

- Primitive recv est bloquante, et Entrée clavier aussi
  - Sous Unix, tout est fichier
    - Utilisation de la primitive **select()** possible ou thread
  - Sous Windows
    - Utilisation de thread

SSL (1)

- SSL (Secure Socket Layer) assure :
  - l'authentification du serveur
  - la confidentialité des données
  - l'intégrité des données
  - (optionnel) l'authentification du client

- Transparent pour l'utilisateur
  - Chiffrement seulement des données
  - Se situe entre la couche TCP et la couche applicative



SSL (2)

- SSLv1 -> juillet 1994 par Netscape (jamais utilisé)

- SSLv2 -> fin 1994, intégré à Netscape Navigator
  en mars 1995 → apparition du **https**

- SSLv3 -> novembre 1995

- TLS (Transport Layer Security)
  -> normalisé par l'IETF
  -> RFC 2246, en 1999
  -> basé sur SSLv3, mais avec de petits changements, donc incompatibilité

  actuellement TLS v1.2 possible

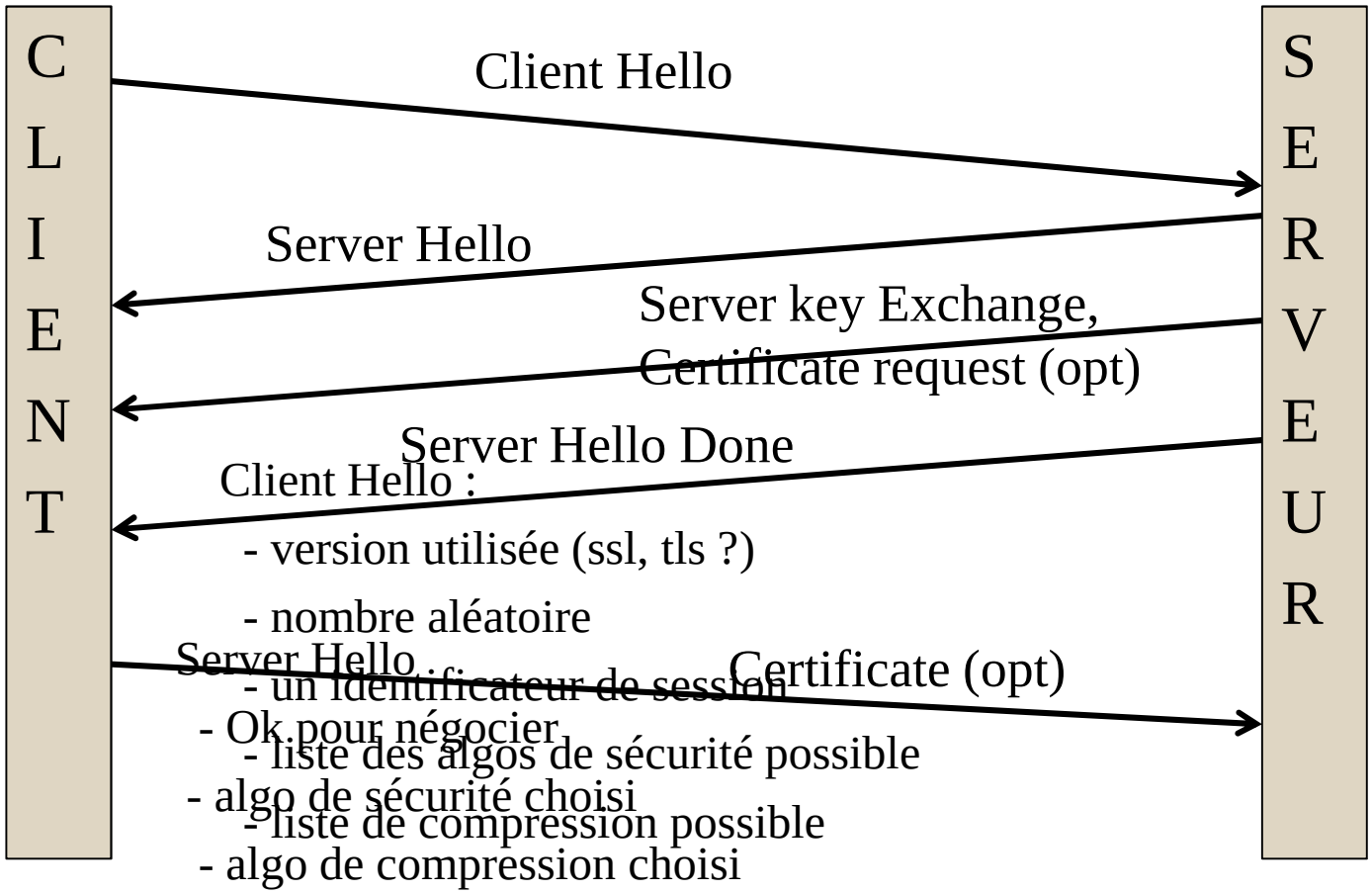
SSL (3)

- SSL est composé de deux étapes
  - SSL Handshake
    - Échange des informations pour le cryptage (longueur de clé, protocoles utilisés, etc..)
    - Utilisation de certificats et de clés publiques pour échanger une clé de session symétrique
  - SSL Record
    - Échange des données cryptées par la clé de session
    - Impossible à décrypter même si les échanges précédents ont été récupérés

Pré-requis serveur : Certificat + clé publique/privée

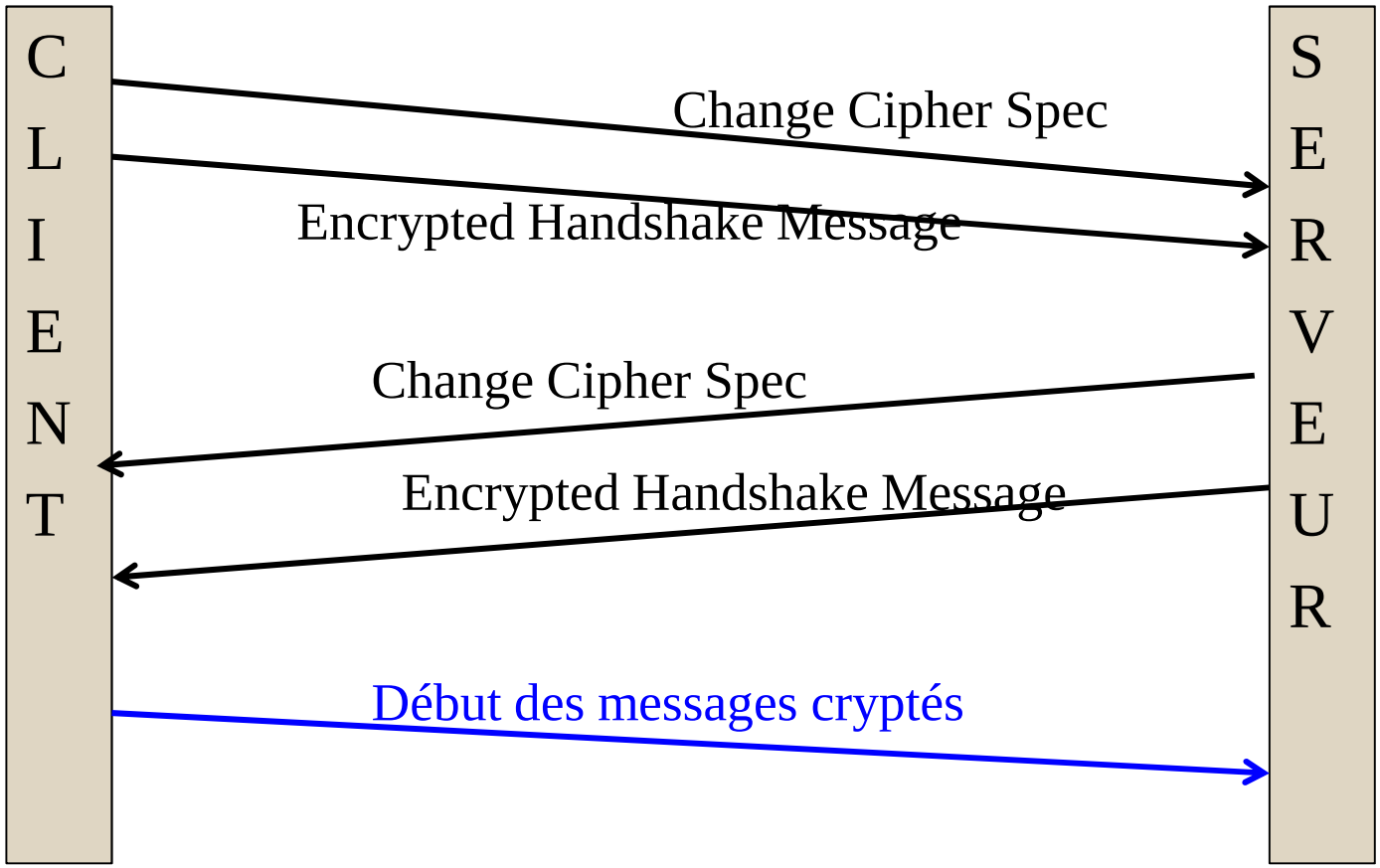
SSL (4)

- Handshake



SSL (5)

- Handshake (suite et fin)



Utilisation ssl en C (1)

- Ouverture de la bibliothèque, création d'un contexte
  - SSL_library_init();
  - OpenSSL_add_all_algorithms();
  - SSL_load_error_string();
- Choix d'une version de ssl
  - SSL_Method *me= SSLv3_{server/client}_method() -> que SSLv3
  - TLSv1 -> que TLSv1 , TLSv1_1 -> que TLSv1.1
  - TLS -> SSLv3, TLSv1, TLSv1.1, TSLv1.2
  - Autre méthode , déprécié (SSLv1_2)
- création d'un contexte
  - SSL_CTX *ctx=SSL_CTX_new (me)

Utilisation ssl en C (2)

- Côté serveur, chargement des certificats
  - SSL_CTX_use_certificate_file(….)
  - SSL_CTX_use_PrivateKey_file(…)
  - SSL_load_error_string();
- Création socket normal avec mise en attente de connexion
- *Mise en relation du contexte ssl avec le client*
  - ssl = SSL_new(ctx);
  - SSL_set_fd(ssl, socket de communication);
- Emission et réception
  - SSL_write(ssl,buffer, taille_buffer);
  - SSL_read(ssl, buffer, taille_buffer);

Utilisation ssl en JAVA(1)

- Utilisation des classes SSLSocket{server}Factory et SSL{Server}Socket
  - **Côté serveur**

```
SSLServerSocketFactory sslserversocketfactory =
        (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
    SSLServerSocket sslserversocket =
        (SSLServerSocket) sslserversocketfactory.createServerSocket(port);
    SSLSocket sslsocket = (SSLSocket) sslserversocket.accept();
```

  - **Côté client**

```
SSLSocketFactory sslsocketfactory = (SSLSocketFactory)
SSLSocketFactory.getDefault();
    SSLSocket sslsocket = (SSLSocket)
sslsocketfactory.createSocket(" nom_serveur", port);
```

Utilisation ssl en JAVA (2)

- Keystore /Truststore
  - Keystore : fichier crypté qui contient clé privé/certificat
  - Truststore : fichier crypté qui contient clé publique et les certificats

  En Java, un client se réfère toujours à un truststore

  Par défaut : *cacerts* → *sous linux /etc/pki/java/cacerts*

- Utilisation d'un autre truststore ou keystore
  - Java –Djavax.net.ssl.trustStore=montrust
    –Djavax.net.ssl.trustStorePassword=passwd nom_programme_client
  - Java –Djavax.net.ssl.keyStore=monkeystore
    –Djavax.net.ssl.keyStorePassword=passwd nom_programme_serveur

- Gestion des clés via le programme : **keytool** (inclus avec la jre)

Exemples

- Exemples dans le répertoire

http://www.isima.fr/~laurenco/prog_socket/

- Socket_c.docx ->> exemple C pour IPv4, IPv6, Csocket
- Socket_java.docx ->> exemple en Java pour Ipv4, Ipv6
- Socket_python.docx ->> exemple en python pour IPv4, Ipv6
- Socket_ssl_c.docs ->> exemple en C pour IPv4 en ssl