Secure Socket Layer (SSL)

Olivier ROUSSEL
olivier.roussel@univ-artois.fr

Le contexte

Insécurité des communications

- À sa création, les utilisateurs d'Internet étaient des personnes parfaitement fréquentables auxquelles on pouvait faire raisonnablement confiance.
- Les principaux problèmes étaient surtout de réussir à communiquer (liaisons bas débit).
- La sécurisation des communications était donc superflue et de toute façon trop coûteuse.
- Pour cette raison, la plupart des protocoles de base de l'Internet sont non sécurisés (IPv4, UDP, TCP,...)
- De nos jours, on ne peut plus faire l'économie de la sécurisation des échanges.

Les problèmes

- Les communications entre deux machines peuvent être écoutées (les données circulent sur des réseaux qui ne sont pas contrôlés par un opérateur de confiance). Si l'on veut échanger des secrets, il faut chiffrer la communication. [confidentialité]
- Même si elles restent secrètes, les données échangées entre deux machines peuvent éventuellement être modifiées par un tiers. [intégrité]
- L'identité d'une personne ou d'une machine peut être usurpée. Avant d'échanger un secret, il faut être sûr que l'on dialogue avec la personne à qui on fait confiance. [authentification]

Les solutions

- Ces problèmes de sécurité se posent de manière évidente avec le développement du web et quand on commence à utiliser l'Internet pour faire du commerce électronique.
- Il fallait sécuriser les transactions sans pour autant remettre en cause le système existant (IPv4, TCP/IP, socket,...) qui a su s'imposer comme une norme.
- La solution est pragmatique : rajouter au dessus des canaux de communication une couche de cryptographie qui soit la plus transparente possible.

SSL/TLS

Principe

Historique

- SSL (Secure Socket Layer) : protocole créé par Netscape en 1993 pour sécuriser l'accès aux pages web (formulaires,...). Correspond aux URL https://...
- L'IETF (Internet Engineering Task Force) est responsable des futurs développements de SSL. Le nouveau protocole s'appelle TLS (Transport Layer Security, RFC 2246)

Objectifs

- SSL/TLS permet de sécuriser une connexion en
  - Authentifiant les serveurs et clients
  - Chiffrant les données échangées
  - Assurant l'intégrité des données
- Une tierce personne ne doit pas pouvoir intercepter un message ou s'immiscer dans un dialogue
- Le protocole doit être suffisamment efficace pour être adapté au web (connexions de faible durée)

Mécanisme

- Le serveur et le client échangent une liste d'algorithmes de cryptographie qu'ils peuvent utiliser et choisissent :
  - Un algorithme à clef publique (pour l'échange de clefs)
  - Un algorithme à clef secrète (plus efficace)
  - Un algorithme de condensé de message (message digest) pour assurer l'intégrité

Mécanisme (2)

- Le serveur envoie sa clef publique certifiée par un tiers de confiance (certificat)
- Le serveur peut demander au client de présenter un certificat
- Le client vérifie le certificat du serveur
- Le client génère une clef secrète (symétrique) aléatoire et la transmet au serveur en la chiffrant avec la clef publique du serveur
- Le serveur accepte la clef secrète (symétrique)
- Les machines échangent des données chiffrées grâce à la clef secrète aléatoire

Remarque

- La programmation du serveur et celle du client sont forcément dissymétriques puisque le serveur **doit** fournir un certificat tandis que cette étape est optionnelle pour le client.

Programmation C/C++

La librairie OpenSSL

- http://www.openssl.org
- Vient s'intercaler au dessus d'un canal de communication (socket, fichier,...) pour réaliser une communication SSL
- Une fois la communication chiffrée établie, on utilise SSL_read et SSL_write pour lire/écrire des informations.
- Pour compiler, il faut ajouter -lssl -lcrypto sur la ligne de commande de g++

Rappel du patron d'un serveur TCP

```
①#include ...
②// var globales et fonctions

void dialogueAvecLeClient(int active) {
④// initier le dialogue
⑤// échanger des données
⑥// clore le dialogue
  close(active);
}

int main() {
③// initialiser la librairie
  passive=socket(...); bind(...); listen(...);
  while(true) {
    active=accept(passive,...);
    if(fork()==0) dialogueAvecLeClient(active); // fils
  }
⑦// finaliser
  close(passive);
}
```


Gestion d'erreur

- Il ne faut pas oublier de gérer les erreurs
- Vérifiez dans les pages de manuel des fonctions ce qu'elles retournent
- Exemple :

```
ret=SSL_accept(ssl);

if (ret<=0)
{
   cerr << "Error SSL_accept: "
        << ERR_error_string(ERR_get_error(),
                            NULL));
   exit(1);
}
```


Initialisation (cas d'un serveur)

- `#include <openssl/ssl.h>`
  `#include <openssl/err.h>`①
- `SSL_CTX *ssl_ctx;`
  contexte SSL (un par programme) ②
- `SSL_load_error_strings();`
  `SSL_library_init();`
  initialiser la librairie ③
- `ssl_ctx=SSL_CTX_new(SSLv23_server_method());`
  initialiser le contexte pour un serveur acceptant les protocoles SSLv2, SSLv3, TLSv1, TLSv1.1 and TLSv1.2 ③
  il faut normalement désactiver les vieux protocoles (SSLv2, SSLv3)

Chargement du certificat et de la clef privée (cas d'un serveur)

- `SSL_CTX_use_certificate_file(ssl_ctx, "fichier.pem", SSL_FILETYPE_PEM)`
  lire le certificat ③
- `SSL_CTX_set_default_passwd_cb(ssl_ctx, pem_passwd_cb)`
  obtenir le mot de passe pour lire le fichier PEM, `pem_passwd_cb` *est un pointeur sur une fonction fournissant ce mot de passe* ③
- `SSL_CTX_use_PrivateKey_file(ssl_ctx, "fichier.pem", SSL_FILETYPE_PEM)`
  lire la clef privée ③
- `SSL_CTX_check_private_key(ssl_ctx)`
  vérifier que la clef privée correspond à la clef publique du certificat ③

Exemple de fonction fournissant le mot de passe

- ②

```
int pem_passwd_cb(char *buf, int size,
                  int rwflag, void *userdata)
{
  strncpy(buf,"Le Mot De Passe", size);
  buf[size-1]='\0';
  return strlen(buf);
}
```

- `rwflag` et `userdata` ne sont pas utiles dans les cas simples.

Établissement de la communication (cas d'un serveur)

- `SSL *ssl=SSL_new(ssl_ctx)`
  structure contenant les paramètres d'une connexion ④
- `SSL_set_fd(ssl,active)`
  attacher le chiffrement à un canal de communication ④
- `SSL_accept(ssl)`
  attendre et procéder à l'échange de clefs (passif),... ④

Lecture/écriture (client et serveur)

- `SSL_write(ssl,buf,len)`
  écriture sur ssl de len octets stockés dans buf ⑤
- `SSL_read(ssl,buf,len)`
  lecture dans buf d'au plus len octets depuis ssl ⑤

Fin de la communication (client et serveur)

À la fin de chaque communication :

- `SSL_shutdown(ssl)` ⑥ *clore la communication*
- `SSL_free(ssl)` ⑥ *libérer la structure une fois la communication rompue*

À la fin du programme :

- `SSL_CTX_free(ssl_ctx)` ⑦ *libérer le contexte une fois le programme terminé*

Rappel du patron d'un client TCP

```
①#include ...
②// var globales

int main() {
③// initialiser la librairie
  active=socket(); connect(active,...);
④// initier le dialogue
⑤// échanger des données
⑥// clore le dialogue (même code que le serveur)
⑦// finaliser (même code que le serveur)
  close(s);
}
```


Initialisation (cas d'un client)

- `#include <openssl/ssl.h>`
  `#include <openssl/err.h>`①
- `SSL_CTX *ssl_ctx;`
  contexte SSL (un par programme) ②
- `SSL_load_error_strings();`
  `SSL_library_init();`
  initialiser la librairie ③
- `ssl_ctx=SSL_CTX_new(SSLv23_client_method());`
  initialiser le contexte pour un client utilisant les protocoles SSL/TLS ③

Établissement de la communication (cas d'un client)

- `SSL *ssl=SSL_new(ssl_ctx)`
  structure contenant les paramètres d'une connexion ④
- `SSL_set_fd(ssl,active)`
  attacher le chiffrement à un canal de communication ④
- `SSL_connect(ssl)`
  procéder à l'échange de clefs (actif),... ④
- il faut aussi vérifier le certificat fourni par le serveur

En C++

- Il est beaucoup plus simple de réaliser les opérations d'entrée/sortie via les flux (opérateurs $<<$ et $>>$). Pour cela, il suffit d'utiliser deux classes ad hoc (non standard).
- `genericbuf` sert d'interface avec le système d'E/S. Il définit les méthodes `basic_read` et `basic_write` qui doivent être redéfinies.
- `SSLBuf` qui redéfinit `basic_read` et `basic_write` pour utiliser `SSL_read` et `SSL_write`.

SSLBuf (dans NetIO.hh)

```
class SSLbuf : public genericbuf
{
private:
  int sock; // connected socket to use
  SSL *ssl; // the SSL struct to use
public:
  SSLbuf(SSL_CTX *ssl_ctx, in sock)
  {
    this->sock=sock;
    ssl=SSL_new(ssl_ctx);
    if (!SSL_set_fd(ssl,Socket)) /* erreur */
  }
  ~SSLbuf()
  {
    SSL_shutdown(ssl);
    close(sock);
    SSL_free(ssl);
  }
```


SSLBuf (suite)

```
  void accept()
  {
    int ret=SSL_accept(ssl);
    if(ret<=0) ... // erreur
  }

  void connect()
  {
    int ret=SSL_connect(ssl);
    if(ret<=0) ... // erreur
  }

  size_t basic_write(const char *buf, size_t count)
  { return SSL_write(ssl,buf,count); }

  size_t basic_read(void *buf, size_t count)
  { return SSL_read(ssl,(char *)buf,count); }
};
```


Utilisation de SSLBuf

- Pour un serveur :

```
// obtention d'une socket active (cas du serveur)
activeSocket=accept(..);
SSLBuf iobuf(ssl_ctx,activeSocket);
iobuf.accept(); // négociation SSL, côté serveur
iostream f(&iobuf);
```

- Pour un client :

```
// obtention d'une socket active (cas du client)
activeSocket=connect(..);
SSLBuf iobuf(ssl_ctx,activeSocket);
iobuf.connect(); // négociation SSL, côté client
iostream f(&iobuf);
```

- Échange de données (client et serveur) :

```
// envoi de données
f << "Nombre=" << i << endl;
// lecture de données
f >> i;
```


Certificat autosigné avec openssl

- On peut créer un certificat autosigné en utilisant par exemple :

```
openssl req -new -x509 -passout
pass:LeMotDePasse -keyout LeFichier.pem
-out LeFichier.pem -days 1000
```

- On peut exporter un certificat au format PEM vers un truststore Java en faisant :

```
openssl x509 -in LeFichier.pem -outform DER
-out export.tmp
keytool -import -alias LeNomDuCertificat
-file export.tmp -keystore
LeFichierTrustStore
```

Programmation Java

Un serveur en Java (depuis Java 1.4)

- `import java.net.*;`
  `import javax.net.*;`
  `import javax.net.ssl.*;`
- `ServerSocketFactory ssocketFactory = SSLServerSocketFactory.getDefault();`
  `ServerSocket ssocket = ssocketFactory.createServerSocket(port);`
  créer la socket SSL
- `Socket socket = ssocket.accept();`
  attendre une connexion et échanger les clefs

Un client en Java

- `import java.net.*;`
  `import javax.net.ssl.*;`
- `SocketFactory socketFactory = SSLSocketFactory.getDefault();`
  `Socket sslSocket = socketFactory.createSocket(hostname, port);`

Entrées/Sorties

```
BufferedReader in=
  new BufferedReader(
    new InputStreamReader(
      sslSocket.getInputStream()));

PrintWriter out=
  new PrintWriter(
    new BufferedWriter(
      new OutputStreamWriter(
           sslSocket.getOutputStream())));
```


En Java

- Ne pas oublier les `close` sur les fichiers
- Ne pas oublier de gérer les exceptions.

Certificat côté serveur

- Le serveur doit obligatoirement avoir un certificat. Pour le préciser au client, une solution possible est de le spécifier par l'intermédiaire de variables d'environnement
- java -Djavax.net.ssl.keyStore=mySrvKeystore -Djavax.net.ssl.keyStorePassword=123456 LeProgramme

Certificat autosigné en Java

- Un certificat autosigné peut être créé par `keytool -genkey -keystore mySrvKeystore -keypass 123456 -keyalg RSA -alias mycert`

Certificat côté client

- Le certificat fourni par le serveur doit être reconnu par le client (sinon exception). Cela signifie qu'il faut utiliser un certificat signé par une CA, ou bien enregistrer côté client le certificat auto-signé comme étant un certificat de confiance.
- Pour un programme de test, on peut par exemple écrire, `java -Djavax.net.ssl.trustStore=mySrvKeystore -Djavax.net.ssl.trustStorePassword=123456 MyApp`
- On peut aussi créer un `TrustManager` spécial qui ne vérifie pas les certificats et l'utiliser (voir http://www-128.ibm.com/developerworks/java/library/j-customssl/).

Annexe : programmation client/serveur Les sockets en TCP

Programmation client/serveur

- Le serveur est la machine à l'écoute des requêtes des clients auxquelles il doit répondre. Il exécute une boucle infinie qui attend une requête d'un client. Il doit pouvoir éventuellement gérer plusieurs clients simultanément en mettant en place des processus ou des threads.
- Le client est la machine qui envoie les requêtes au serveur et réceptionne les réponses de ce dernier. C'est lui qui prend l'initiative de contacter le serveur.

Origine des sockets

- API introduite en 1982 dans la version BSD d'Unix
- Orientation Unix : une socket va se comporter comme un fichier (read et write sont directement utilisables sur des sockets TCP)
- Cette API n'est pas restreinte aux seules communications TCP/IP (socket unix,...)
- Standard de fait
- Cette API devait représenter différents types de communications, dans un langage (C) qui ne dispose pas des notions d'objet, d'héritage et de polymorphisme. Ces notions sont donc "simulées" en utilisant des pointeurs.

Le terme socket

- Socket signifie "prise" et symbolise ici la prise téléphonique murale qui est le prérequis pour toute conversation téléphonique.

Différents types de sockets

- Socket passive : socket d'un serveur qui est en attente d'une connexion d'un client. Cette socket n'envoie et ne réceptionne aucune donnée car il y n'y a aucun correspondant désigné à l'autre bout de la ligne. [≈ téléphone raccroché]. Une file d'attente (listen queue) permet de faire "patienter" les connexions le temps que le serveur puisse les prendre en compte [≈ intermède musical]
- Socket active : socket qui permet la communication avec un correspondant à l'autre bout de la ligne [≈ téléphone décroché]

Différents types de sockets

- AF_UNIX (Address Family UNIX)
  Socket unix qui permet des communications locale à une machine
- AF_INET (Address Family Internet (IPv4))
  AF_INET6 (Address Family Internet 6 (IPv6))
  - SOCK_STREAM : protocole TCP
  - SOCK_DGRAM : protocole UDP
  - SOCK_RAW : datagramme IP
- et aussi :
  - AF_IPX (IPX - Novell protocols)
  - AF_NETLINK (Kernel user interface device)
  - AF_X25 (ITU-T X.25 / ISO-8208 protocol)
  - AF_AX25 (Amateur radio AX.25 protocol)
  - AF_ATMPVC (Access to raw ATM PVCs)
  - AF_APPLETALK (Appletalk)
  - AF_PACKET (Low level packet interface)
- Les constantes PF_xxx (Protocol Family) sont équivalentes à AF_xxx et obsolètes.

Rappels sur TCP/UDP

- TCP permet de transmettre un flux d'octets [≈ communication téléphonique vocale]
  - mode connecté : avant tout envoi de donnée, il y a négociation préalable de divers paramètres avec le destinataire (triple poignée de main : trois segments sont échangés avant l'envoi du premier octet)
  - envoi d'un flux d'octets : le programmeur lit et écrit une suite d'octets. C'est la couche TCP qui se charge de découper le message en datagrammes
  - garantie sur la transmission : sauf catastrophe, les octets arrivent à destination (retransmission en cas d'absence d'accusé de réception), dans le bon ordre et sans altération.
  - inutilisable avec le broadcast ou le multicast.
  - surcoût significatif du protocole (en-tête plus gros qu'en UDP, accusés de réception, ...)

Rappels sur TCP/UDP

- UDP permet d'envoyer des datagrammes [≈ envoi de SMS]
  - mode non connecté : on peut envoyer des données sans négociation préalable avec le destinataire
  - envoi de datagrammes : le programmeur doit se charger de découper son message en datagrammes
  - pas de garantie sur la transmission : les messages peuvent se perdre, arriver dans le désordre et être altérés. Le programmeur doit gérer cela.
  - utilisation possible pour le broadcast ou le multicast.
  - surcoût du protocole minimal
- On utilise UDP quand :
  - on doit faire du broadcast ou du multicast
  - ou quand les questions et les réponses sont courtes, auquel cas on peut gérer facilement les pertes de messages sans avoir la lourdeur de TCP

Serveur TCP en C/C++

Schéma de création d'une socket passive :

- Appel de socket() pour spécifier les protocoles à utiliser et obtenir un descripteur de fichier (entier servant d'identifiant) [≈ branchement d'une prise murale]
- Appel de bind() pour fixer l'adresse IP et le port sur lequel il faut attendre les connexions. Possibilité d'écouter sur toutes les adresses IP (INADDR_ANY) de la machine et de laisser le système choisir un port libre (0) [≈ attribution d'un numéro de téléphone]

Serveur TCP en C/C++

Schéma de création d'une socket passive (suite) :

- Appel de listen() pour fixer la taille de la file d'attente des clients [≈ nombre de coups de fil qui peuvent être mis en attente]
- Appel de accept() pour attendre l'appel d'un client (appel bloquant). [≈ patienter devant le téléphone] Quand un client se connecte, accept() crée une socket active permettant la communication avec le client et renvoie le descripteur de fichier de cette socket active [≈ décrocher et transmettre la communication à la personne qui va répondre]

Client TCP en C/C++

Schéma de création d'une socket active

- Appel de socket() pour spécifier les protocoles à utiliser et obtenir un descripteur de socket [≈ branchement d'une prise murale]
- *OPTIONNEL : appel de bind() pour fixer l'adresse IP et le port à partir duquel on effectue la connexion. [≈ choisir le numéro de téléphone à partir duquel on appelle]*
- Appel de connect() pour identifier le serveur auquel on veut se connecter (IP+port) [≈ composer le numéro du serveur]

Fonctions communes en TCP

- send() ou write() : écriture de données sur une socket
- recv() ou read() : lecture de données sur une socket
- getsockopt() : obtenir des informations sur la socket (le numéro IP du client par exemple)
- setsockopt() : modifier certains paramètres de la socket
- close() : fermer la communication et libérer la socket

Little and big endians (petit et grand boutisme)

D'un processeur à l'autre, les entiers stockés sur plusieurs octets ne sont pas forcément représentés de la même manière :

- Little endian (petit boutisme) : le premier octet de l'entier est le moins significatif (x86) : on commence par le petit bout (little end)
- Big endian (grand boutisme) : le premier octet de l'entier est le plus significatif (680x0, PowerPC, ordre adopté sur le réseau) : on commence par le grand bout (big end)
- Il existe d'autres conventions (PDP)

On retrouve ces différences de conventions dans les différentes langues :

- Little endian : vierundzwanzig, 24 (dans les langues écrites de droite à gauche), 31/12/2050
- Big endian : vingt-quatre, twenty-four, 24 (dans les langues écrites de gauche à droite), 2050-12-31

Little and big endians

Exemple : représentation mémoire du nombre 0x01020304=$1*256^3+2*256^2+3*256^1+4*256^0$

- Little endian :

04	03	02	01
0	*1*	*2*	*3*

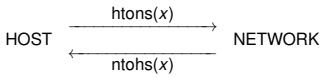
- Big endian :

01	02	03	04
0	*1*	*2*	*3*

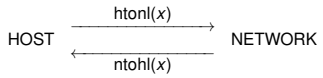
Échange d'entiers

Pour échanger des entiers, il faut bien sûr adopter la même représentation. Pour cela, on convertit systématiquement tout entier sur plusieurs octets en utilisant la convention adoptée sur Internet (big endian). On utilise pour cela les fonctions hton (Host TO Network) et ntoh (Network TO Host).

- Entier sur 2 octets (short)



- Entiers sur 4 octets (long)



Autres problèmes d'échanges de données

Différences de codage des caractères :

- ASCII
- latin1 (iso-8859-1) et autres codages
- Unicode : UTF-8, UTF-16, UTF-32
- ...

Différences de représentation de la fin de ligne :

- Unix : LF
- Mac OS jusque Système 9 : CR
- Windows : CR LF
- ...

Serveur TCP en IPv4 (version C/C++)

Squelette du serveur

- les includes

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

- des variables

```
// descripteur de fichier
int passiveSocket;
// couple (adresse,port) en IPv4
struct sockaddr_in servAddr;
```

- des constantes

```
// le port sur lequel on écoute
const int serverPort=2000;
```


Le serveur : appel à socket()

```
// création de la prise murale
passiveSocket=socket(AF_INET,SOCK_STREAM,0);
if(passiveSocket<0)
{
  perror("erreur socket");
  exit(1);
}
```


Le serveur : appel à bind()

```
// mettre à 0 tous les octets de servAddr
memset(&servAddr,0,sizeof(servAddr));
// initialiser les champs
servAddr.sin_family=AF_INET; // IPv4
// adresse sur laquelle on écoute
servAddr.sin_addr.s_addr=htonl(INADDR_ANY);
// port sur lequel on écoute
servAddr.sin_port=htons(serverPort);

// assignation de l'adresse IP et du numéro de port
if(bind(passiveSocket,
        (struct sockaddr *)&servAddr,
        sizeof(servAddr))
    <0)
{
  perror("erreur bind");
  exit(1);
}
```


Le serveur : appel à listen()

```
// fixer la taille de la file d'attente
const int tailleFileDAttente=5;
if(listen(passiveSocket,tailleFileDAttente)<0)
{
  perror("erreur listen");
  exit(1);
}
```


Le serveur : appel à accept()

```
while (true) // boucle infinie
{
  struct sockaddr clientAddr;
  socklen_t clientAddrLength;
  int activeSocket;
  pid_t child;
  clientAddrLength=sizeof(clientAddr);

  // attendre une communication (appel bloquant)
  // retourne une socket active et l'adresse du client
  activeSocket=accept(passiveSocket,
                      &clientAddr,&clientAddrLength);
  if(activeSocket<0)
  {
    perror("erreur accept");
    exit(1);
  }
  // à suivre
```

Le serveur : appel à fork()

```
if((child=fork())==0)
{
  /* Processus fils */
  close(passiveSocket);

  printf("Connexion de %s:%d\n",
         inet_ntoa(((struct sockaddr_in *)&clientAddr)
                   ->sin_addr),
         ntohs(((struct sockaddr_in *)&clientAddr)
                   ->sin_port));
  dialoguerAvecLeClient(activeSocket);
  printf("Fin de connexion\n");
  close(activeSocket);
  exit(0);
}
else
```


Le serveur : fork() suite

```
    if(child>0)
    {
      /* Processus père */
      close(activeSocket);
      /* Attendre la connexion suivante */
    }
    else
    {
      perror("fork");
      exit(1);
    }
  } // while(true)
```


Zombies

- quand un père crée un processus fils, il doit attendre la fin de ce fils et prendre connaissance du code de retour de ce fils (valeur retournée par exit()).
- tant que le père n'a pas pris connaissance de ce code, un fils qui s'est terminé est dans l'état Zombie (Z). Il ne s'exécute plus, mais ne peut pas être effacée de la table des processus pour permettre au père de récupérer son code de retour avec l'une des fonctions wait().
- comme on ne sait pas quand le fils se termine, il faut utiliser des signaux pour demander à être prévenu (et interrompu) quand un fils se termine.

Attente des fils

```
void waitChild(int signum, siginfo_t *siginfo,
               void *ucontext) {
  int status;
  // un seul signal peut correspondre à l'arrêt de
  // plusieurs fils. On récupère le résultat de chaque
  // fils déjà terminé, sans attendre
  while(waitpid(-1,&status,WNOHANG)>0);
}

int main() {
  // mise en place du gestionnaire
  struct sigaction handler;

  handler.sa_sigaction=waitChild;
  sigemptyset(&handler.sa_mask);
  handler.sa_flags=SA_SIGINFO|SA_RESTART;

  sigaction(SIGCHLD,&handler,NULL);
  ...
```


Client TCP en IPv4 (version C/C++)

Le client : appel à socket()

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <netdb.h>

int main() {
  int activeSocket; // descripteur de fichier

  // création de la prise murale
  activeSocket=socket(AF_INET,SOCK_STREAM,0);
  if(activeSocket<0)
  {
    perror("erreur socket");
    exit(1);
  }
```


Le client : obtenir l'adresse du serveur

ATTENTION : il ne faut plus utiliser `gethostbyname()` mais utiliser `getaddrinfo()` à la place (voir plus loin)

```
struct sockaddr_in servAddr;
char *hostname="serveur.dom.fr"; // ou "1.2.3.4"

// tout mettre à 0
memset(&servAddr,0,sizeof(servAddr));
servAddr.sin_family=AF_INET;
// fournir le port sur lequel le serveur écoute
servAddr.sin_port=htons(serverPort);
// récupérer l'adresse du serveur auprès du DNS
struct hostent *ip=gethostbyname(hostname);
if(ip==NULL) {
  herror("échec de la résolution de nom: ");
  exit(1);
}
// copie de l'adresse
memcpy(&servAddr.sin_addr,ip->h_addr,ip->h_length);
```


Le client : appel à connect()

```
  // établir la connexion avec le serveur
  if(connect(activeSocket,(struct sockaddr *)&servAddr,
             sizeof(servAddr))<0)
  {
    perror("erreur connect");
    exit(1);
  }

  dialoguerAvecLeServeur(activeSocket);

  close(activeSocket);
} // fin main
```


Client/Serveur TCP en IPv6 (version C/C++)

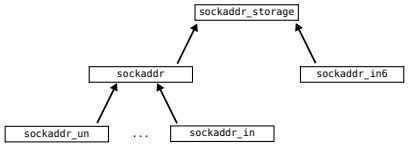
Les changements mineurs

IPv4	IPv6
`struct sockaddr`	`struct sockaddr_storage`
`struct sockaddr_in`	`struct sockaddr_in6`
`socket(AF_INET, SOCK_STREAM, 0)`	`socket(AF_INET6, SOCK_STREAM, 0)`

`// adresse quelconque`
`struct sockaddr_in addr;`
`addr.sin_addr.s_addr =htonl(INADDR_ANY);`
	`// adresse quelconque`
	`struct sockaddr_in6 addr;`
	`addr.s_addr=in6addr_any;`

Les structures IPv6 sont plus grosses que les structures IPv4. Une variable de type `struct sockaddr_storage` peut contenir soit une `sockaddr_in6`, soit une `sockaddr_in`.

Pseudo diagramme de classe



Résolution de nom

- à la place de gethostbyname(), il faut utiliser

```
int getaddrinfo(const char *nodename,
                const char *servname,
                const struct addrinfo *hints,
                struct addrinfo **res);

void freeaddrinfo(struct addrinfo *res);

const char *gai_strerror(int errcode);
```

- quand getaddrinfo() retourne 0, il fournit une liste chaînée d'adresses (IPv6 ou IPv4) dans res (pointeur sur la tête). Cette liste doit être libérée par freeaddrinfo (). Quand getaddrinfo() retourne une valeur non nulle, c'est un code d'erreur qui peut être déchiffré par gai_strerror().

getaddrinfo()

```
int getaddrinfo(const char *nodename,
                const char *servname,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

- `nodename` est le nom de la machine. S'il est NULL, les réponses contiendront l'adresse INADDR_ANY si hints.ai_flags contient le bit AI_PASSIVE (pour un serveur), et l'adresse de loopback sinon
- `servname` est le nom d'un **service** (par exemple ssh,ftp,http,...) ou un numéro de port sous forme de chaîne de caractères. S'il est NULL, les réponses contiendront un numéro de port égal à 0.
- Seul un des paramètres `nodename` ou `servname` peut être NULL.

getaddrinfo() suite

- `hints` permet d'imposer de filtrer les réponses grâce aux champs ai_flags, ai_family et ai_socktype (NULL si aucun filtre). man getaddrinfo pour le détail des flags.
- les réponses (et hints) sont stockées dans la structure suivante

```
struct addrinfo {
   int ai_flags; /* AI_PASSIVE, AI_CANONNAME, ... */
   int ai_family; /* AF_INET, AF_INET6, AF_UNSPEC */
   int ai_socktype; /* SOCK_STREAM, SOCK_DGRAM */
   int ai_protocol; /* 0 en général */
   size_t ai_addrlen; /* la taille de ai_addr */
   char *ai_canonname; /* le FQDN */
   struct sockaddr *ai_addr; /* l'adresse binaire */
   struct addrinfo *ai_next; /* noeud suivant */
};
```


getaddrinfo() : exemple côté serveur

```
struct addrinfo hints, *resultList;
const char *port="10000";

memset(&hints, 0, sizeof hints) ;
hints.ai_family=AF_UNSPEC; // IPv4 ou IPv6
hints.ai_socktype=SOCK_STREAM;
// on veut une socket passive
hints.ai_flags=AI_PASSIVE;

err = getaddrinfo(NULL,port,&hints,&resultList);
if(err)
{
   fprintf(stderr,"getaddrinfo: %s\n",
           gai_strerror(err));
   exit(1);
}
```


getaddrinfo() : exemple côté serveur

```
n=0;
for(struct addrinfo *res=resultList;res!=NULL;
    res=res->ai_next)
{
   sock[n]=socket(res->ai_family,res->ai_socktype,
                  res->ai_protocol);
   if(sock[n]<0) {
      perror("socket");
      exit(1);
   }
   if(bind(sock[n],res->ai_addr,res->ai_addrlen)<0) {
      perror("bind");
      exit(1);
   }
   ...
   n++;
}
freeaddrinfo(resultList);
```


Remarques

- Sur une machine où IPv4 et IPv6 sont disponibles, on obtient dans l'exemple précédent deux adresses sur lesquelles il faut écouter (une IPv4 et une IPv6). Il faudra utiliser select() pour determiner laquelle des sockets reçoit une connection avant de faire un accept().
- Sur la plupart des systèmes IPv6, on peut se contenter d'être à l'écoute sur l'adresse IPv6 (les connexions IPv4 passeront par la pile IPv6) à condition d'autoriser la socket à recevoir les flux IPv4 et IPv6 par `bool b=false; setsockopt(sock, IPPROTO_IPV6, IPV6_V6ONLY, (char *)&b, sizeof(b))`.
- Côté client, on appelle getaddrinfo(), on parcourt la liste chaînée en essayant de faire un connect() et on quitte la boucle dès qu'on a réussi à établir une connexion avec le serveur.

Dialogue entre le client et le serveur

Brefs rappels sur les entrées/sorties

- On peut utiliser différentes fonctions pour les E/S
  - niveau Unix (le plus bas) : open(), read(), write(), close()
  - niveau C : fopen(), fread(), fwrite(), fprintf(), fscanf(), fclose()
  - niveau C++ (le plus haut) : <<, >>, f.read(), f.write(), f.close()

  De manière générale, les fonctions de haut niveau doivent être préférées aux fonctions de bas niveau.
- On peut transmettre des données sous forme
  - texte (simple mais lent)
  - binaire (rapide mais attention à little/big endian)

Écriture et lecture sur une socket TCP (niveau Unix)

- Une socket SOCK_STREAM se comporte comme un fichier UNIX identifié par un descripteur s (numéro entier)
- Pour écrire n octets à partir de l'adresse buffer
  - send(s,buffer,n,flags) /* socket SOCK_STREAM uniquement, flags=0 en général */
  - write(s,buffer,n) /* tous fichiers UNIX */
- Pour lire au plus n octets et les placer à partir de l'adresse buffer
  - recv(s,buffer,n,flags) /* socket SOCK_STREAM uniquement, flags=0 en général */
  - read(s,buffer,n) /* tous fichiers UNIX */
- Les appels renvoient le nombre d'octets écrits ou lus.

Exemple d'écriture au niveau Unix

- Écriture d'un entier en ascii :

```
char buffer[MAX];
snprintf(buffer,MAX,"%d\n",i);
write(s,buffer,strlen(buffer));
```

- Écriture d'un entier en binaire :

```
long int l,tmp;
tmp=htonl(l);
write(s,&tmp,sizeof(tmp));
```


Exemple de lecture au niveau Unix

- Lecture d'un entier en ascii :

```
char buffer[MAX];
len=read(s,buffer,MAX);
/* il faut gérer les cas où on lit trop ou pas
   assez d'octets */
sscanf(buffer,"%d",&i);
```

- Lecture d'un entier en binaire :

```
long int l,tmp;
len=read(s,&tmp,sizeof(tmp));
/* il faut gérer les cas où on lit trop peu
   d'octets ou utiliser
   recv(s,&tmp,sizeof(tmp),MSG_WAITALL)
*/
l=ntohl(tmp);
```


Lecture/écriture au niveau C (avec tampon)

- On peut créer un fichier du type FILE * à partir d'une socket par

```
FILE *f=fdopen(s,mode)
```

avec mode= "w" ou "r" (il faut créer un fichier pour la lecture et un autre pour l'écriture)

- On peut ensuite directement utiliser

```
fprintf(f, "%d\n",i); // envoi de texte
fscanf(f,"%d",&i); // réception de texte
fwrite(buffer,n,1,f); // envoi binaire
fread(buffer,n,1,f); // réception binaire
fflush(f); // vider le tampon
           // (forcer la transmission)
```

et autres fonctions sur les FILE *

Interface entre fichiers C++ et socket

- Le standard C++ ne permet pas de construire un iostream à partir d'un descripteur de fichier.
- Avec g++, on peut utiliser la classe non standard `__gnu_cxx::stdio_filebuf<char>`.
- Exemple avec g++ :

```
#include <ext/stdio_filebuf.h>
// obtenir une socket active
activeSocket=accept(...);
// créer un objet filebuf à partir de la socket active
__gnu_cxx::stdio_filebuf<char> fbuf(activeSocket,
                                    ios::in|ios::out);
// créer un fichier en lect./ecr. à partir du filebuf
iostream f(&fbuf);

// envoi
int i=5;
f << "Ligne "<< i << endl;

// reception
string s;
getline(f,s); // lire une ligne complète
f >> s >> i; // lire un mot, puis un entier
```


De la nécéssité du flush

- Chaque fois qu'on utilise des E/S avec tampon (buffer), le système attend d'avoir suffisamment de données pour qu'il soit rentable de les envoyer.
- Les données du programme peuvent donc rester indéfiniment dans le tampon si on ne force pas leur envoi.
- Il est donc nécessaire de forcer l'envoi des données en vidant le tampon par un appel à flush()
  - C : fflush(file);
  - C++ : f.flush(); // ou bien f << flush;
  - C++ : f << endl; // envoi d'une fin de ligne et flush
  - Java : out.flush();

Protocole entre client et serveur

- Le programmeur doit définir comment le client et le serveur dialoguent. C'est ce qu'on appelle le protocole.
- De manière évidente, quand un client envoie un message, le serveur doit être prêt à le lire.
- De la même manière, quand un serveur envoie un message, le client doit être prêt à le lire.
- Les E/S du client et du serveur doivent donc se correspondre. C'est au programmeur de le garantir.
- Si le client et le serveur attendent tous les deux de recevoir des données, il y a interblocage.
- Ne pas confondre les lectures/écritures sur la socket (réception/envoi de données) avec les lectures/écritures sur l'entrée/sortie standard (clavier/écran) !

Client/Serveur TCP en Java

Client/Serveur TCP en Java

- Le langage Java inclut en standard les classes permettant de réaliser une connexion TCP
  import java.net.*;
- On utilise les classes
  - ServerSocket pour le serveur TCP
  - Socket pour le client TCP

Serveur TCP en Java

- `ServerSocket passiveSocket=new ServerSocket(port);`
  cette classe encapsule les appels socket,bind et listen de C. Il existe diverses versions du constructeur pour spécifier les paramètres de bind et listen
- `Socket activeSocket=passiveSocket.accept();`
  équivalent du accept de C
- `activeSocket.getInputStream()` et `activeSocket.getOutputStream()` renvoient les flux pour lire/écrire sur la socket
- La méthode close() permet de fermer les sockets

Utilisation des threads

- Pour créer un processus léger pour gérer chaque connexion :

```
while (true)
{
  Socket activeSocket=passiveSocket.accept();
  Thread t=new MyThread(activeSocket);
  t.start();
}
```

- La classe MyThread doit hériter de la classe Thread et redéfinir la méthode run() pour qu'elle effectue le travail souhaité. Le constructeur de MyThread doit prendre en charge les paramètres.

Gestion des flux

- Obtention de flux de type texte

```
BufferedReader in=new BufferedReader(
  new InputStreamReader(
    activeSocket.getInputStream()));

String line=in.readLine();

PrintWriter out=
  new PrintWriter(
    activeSocket.getOutputStream());

out.println("message ");
```


Client TCP en Java

- `Socket activeSocket=new Socket(host,port);`
  encapsule les appels socket et connect de C
- Ne pas oublier de fermer les sockets

Divers points en Java

- Il faut bien sûr prendre soin de gérer les diverses exceptions qui peuvent survenir
- La classe InetAddress permet si nécessaire de convertir un nom de machine en adresse IP ou inversement
- Voir la documentation Java pour plus d'informations

Serveur d'echo en Java 1/3

```
import java.io.*;
import java.net.*;
import java.util.*;

class ServeurEcho extends Thread {
  final static int port = 5000;
  Socket s;

  ServeurEcho(Socket s) {
    this.s = s;
  }
```


Serveur d'echo en Java 2/3

```
  public void run() {
    try {
      BufferedReader in = new BufferedReader(
          new InputStreamReader(s.getInputStream()));
      PrintWriter out = new PrintWriter(s.getOutputStrea

      String line;

      while((line=in.readLine())!=null) {
        out.println("echo : "+line);
        out.flush();
      }

      s.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
```


Serveur d'echo en Java 3/3

```
  public static void main(String[] args) {
    try {
      ServerSocket passiveSocket = new ServerSocket(port

      while (true) {
        Socket activeSocket = passiveSocket.accept();
        ServeurEcho s = new ServeurEcho(activeSocket);
        s.start();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```


Client d'echo en Java 1/2

```
import java.io.*;
import java.net.*;

class Client {
  public static void main(String[] args) {
    try {
      Socket s = new Socket("nom.du.serveur",5000);

      BufferedReader in = new BufferedReader(
          new InputStreamReader(s.getInputStream()));
      PrintWriter out = new PrintWriter(s.getOutputStrea
```


Client d'echo en Java 2/2

```
      String ligne;

      out.println("Hello");
      out.flush();

      ligne = in.readLine();
      System.out.println("Le serveur a répondu : "+ligne

      s.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```