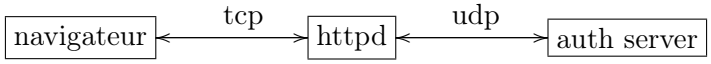
TP socket : un serveur HTTP avec autorisation

1 L'objectif

L'objectif final est de réaliser un serveur web qui demande l'autorisation.



Le navigateur se connecte (socket en mode connecté) au serveur httpd, envoie une requête http, attend une réponse http et l'affiche.

Le serveur httpd attend des demandes de connexions et crée une activité (un thread) pour chacune d'elles. Cette activité lit la requête, la décode, vérifie l'autorisation, la traite, et renvoie une réponse.

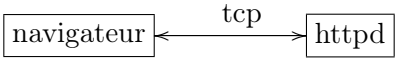
La requête envoyée par le navigateur est un texte de la forme :

```
GET /README HTTP/1.1
Host: posets.enseeiht.fr

```

Le mot `GET` identifie la méthode (obtention d'une page), et le chemin d'accès qui suit est la partie locale de l'URL, laquelle correspond, pour notre serveur, à un nom de fichier.

1.1 Étape 1 : httpd sans autorisation



— La seule requête HTTP acceptée est GET filename
— Le traitement consiste à renvoyer le contenu du fichier s'il existe, ou une erreur sinon.
— Les deux formats de réponse sont :

```
HTTP/1.1 200 Ok
Server: le mien
Content-Type: text/plain

contenu du fichier
```

ou

```
HTTP/1.1 404 Not Found
Server: le mien
Content-Type: text/plain

File not found
```

À faire : compléter `httpd.c` et tester :

```
./httpd 8040 0   # 8040 = port sur lequel tourne le serveur httpd
```

puis dans un navigateur, charger l'url `http://localhost:8040/Makefile` ou `http://localhost:8040/foo`.

1.2 Étape 2 : Le mécanisme d'autorisation

Le serveur httpd doit maintenant avoir l'autorisation de répondre au navigateur. Pour cela, il diffuse un message en broadcast et attend une réponse. Si la réponse arrive et est positive, elle reste valide pendant `AUTH_LIFETIME` secondes. Si elle est négative ou s'il n'y a pas de réponse après un délai de garde (`AUTH_CLIENT_TIMEOUT`) :

```
HTTP/1.1 403 Forbidden
Server: le mien
Content-Type: text/plain

Forbidden access.
```

Noter que httpd est serveur du navigateur, et client du service d'authentification : un processus n'est pas forcément uniquement client ou uniquement serveur.

À faire : compléter `auth_client.c` et ajouter dans `httpd.c` les appels à `auth_init` et `auth_check`.

Pour tester :

```
./auth_server 9000 &  # 9000 = port du serveur d'autorisation
./httpd 8040 9000    # 8040 = port sur lequel tourne le serveur httpd
```

Note : vu que le serveur d'autorisation est contacté en broadcast, il ne faut pas que tous les étudiants (d'un même réseau) utilisent le même numéro de port 9000. Le risque est en effet d'avoir plusieurs réponses pour une seule demande d'autorisation.

2 Codes fournis

2.1 `common.h/c`

Voir dans `common.h` pour plus d'informations.

```
/* Formatted write on a file descriptor. Works like printf. */
void writef(int fd, const char *fmt, ...);

/* Read a http request from the file descriptor fd and return the first line of the request. */
char *read_request(int fd);

/* Parse the request, returns method and url. */
void parse_request(char *line, int *method, char **filename);

/* Return the mimetype for the given filename. */
char *find_mimetype(char *filename);
```

2.2 `auth_server.c`

Une implantation complète d'un serveur d'autorisation est fournie dans `auth_server.c`. Ce serveur est assez mesquin, ignorant certaines des requêtes (comme si elles avaient été perdues) et répondant arbitrairement oui ou non aux autres. Le format des messages à échanger est décrit dans `auth.h` : le demandeur envoie un `Auth_Message` contenant `AUTH_QUERY` et le serveur répond éventuellement un `Auth_Message` contenant `AUTH_ACK` ou `AUTH_NACK`.

3 Rappels

3.1 Posix thread

```
pthread_t t;
pthread_create(&t, NULL, foo, arg);
```

Crée une nouvelle activité concurrente pour exécuter `foo(arg)`. `arg` est un pointeur vers les données fournies au thread, on fera attention à fournir un pointeur différent pour chaque thread (`malloc`).

3.2 E/S Unix

3.2.1 Ouverture d'un fichier `open`

```
int fd;
fd = open (filename, O_RDONLY);
```

3.2.2 Lecture/écriture `read`/`write`

```
int read (int fd, void *buf, int nbyte);
int write (int fd, const void *buf, int nbyte);
void writef(int fd, const char *fmt, ...); /* (cf common.h). */
```

L'appel système `read` lit sur le descripteur au plus `nbyte` octets et les range à l'adresse `buf`. Il renvoie le nombre d'octets effectivement lus, ou 0 si la « fin de fichier » a été atteinte (il n'y aura plus rien à lire : socket fermé ou fichier complètement lu), ou -1 en cas d'erreur.

L'appel système `write` écrit sur le descripteur au plus `nbyte` octets rangés à l'adresse `buf`. Il renvoie le nombre d'octets effectivement écrits, ou -1 en cas d'erreur.

La fonction `writef` fournie dans `common.c` permet d'effectuer des écritures formatées à la printf sur un descripteur de fichier.

Copier le contenu d'un fichier sur un descripteur `fdest` : (il manque le contrôle d'erreur sur `read` et `write`)

```
int n; char buf[256];
int fic = open(filename, O_RDONLY);
if (fic == -1) { perror("Can't open file"); ... }
while ((n = read(fic, buf, 256)) > 0) { write(fdest, buf, n); }
close(fic);
```

3.3 Socket en mode connecté

Un socket est un point de communication par lequel le processus peut émettre ou recevoir des informations vers ou en provenance d'un autre socket.

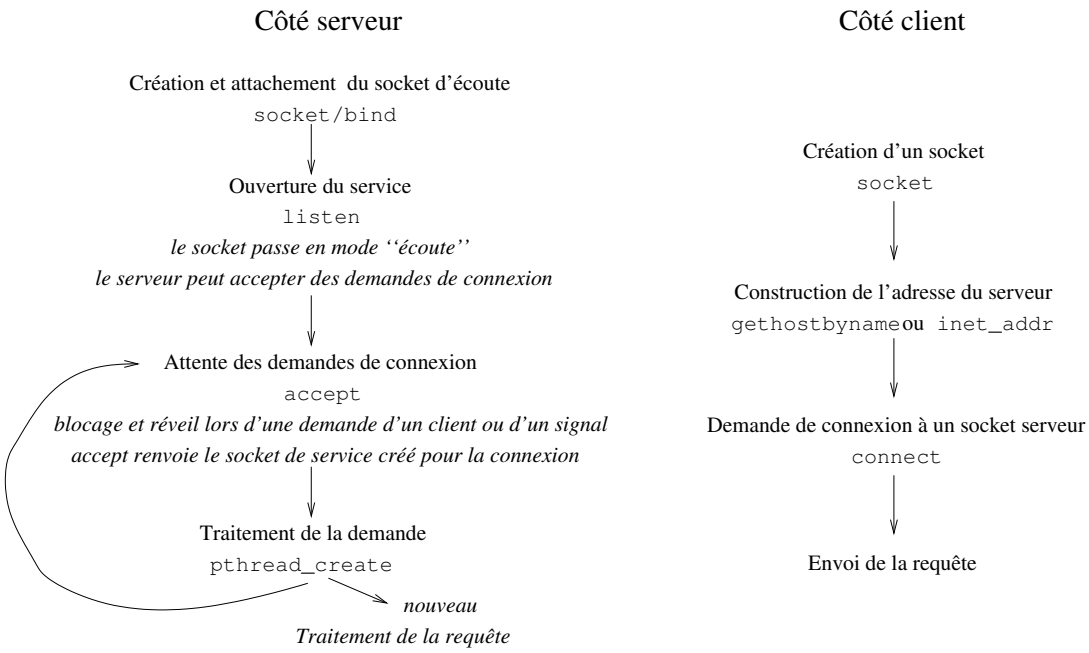


FIGURE 1 – Socket en mode connecté

3.3.1 Le type `sockaddr_in`

Une adresse de socket dans la famille Internet est définie par :

```
struct sockaddr_in {
        short   sin_family;       /* la famille de protocole */
        u_short sin_port;         /* numéro de port */
        struct  in_addr sin_addr; /* adresse IP de la machine */
        char    sin_zero[8];      /* remplissage pour faire 16 octets */
};
```

L'adresse IP d'une machine (type `struct in_addr`) est en fait 4 octets, qu'on écrit généralement sous la forme 147.127.64.7.

3.3.2 Côté client

On utilise des sockets dans la famille de protocole Internet, de type stream (fiable, fifo, bi-directionnel). Dans la suite, `serverhost` est le nom de la machine que l'on veut contacter, `port` est le numéro du port sur cette machine :

```
int sc;
struct hostent *sp;
struct sockaddr_in sins;
/* Obtention d'information au sujet de la machine 'serverhost' */
sp = gethostbyname (serverhost);
if (sp == NULL) {
    fprintf (stderr, "gethostbyname: %s not found\n", serverhost);
    exit (1);
}
```

```
/* Création d'un socket Internet de type stream (fiable, bi-directionnel) */
sc = socket (AF_INET, SOCK_STREAM, 0);
if (sc == -1)
    err (1, "socket failed");

/* Remplissage de la structure 'sins' avec la famille de protocoles Internet,
 * le numéro IP de la machine à contacter et le numéro de port. */
sins.sin_family = AF_INET;
memcpy (&sins.sin_addr, sp->h_addr_list[0], sp->h_length);
sins.sin_port = htons (port);
/* Tentative d'établissement de la connexion. */
if (connect (sc, (struct sockaddr *)&sins, sizeof(sins)) == -1)
    err (1, "connect failed");
```

3.3.3 Côté serveur

```
struct sockaddr_in soc_in;
int val;
int ss;  /* socket d'écoute */

/* socket Internet, de type stream (fiable, bi-directionnel) */
ss = socket (AF_INET, SOCK_STREAM, 0);

/* Force la réutilisation de l'adresse si non allouée */
val = 1;
setsockopt (ss, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));

/* Nomme le socket: socket inet, port PORT, adresse IP quelconque */
soc_in.sin_family = AF_INET;
soc_in.sin_addr.s_addr = htonl (INADDR_ANY);
soc_in.sin_port = htons (port);
bind (ss, &soc_in, sizeof(soc_in));

/* Prépare le socket à la réception de demande de connexions */
listen (ss, 5);

while (1) {
    struct sockaddr_in from;
    int len;
    int f;

    /* Accepte une connexion.
     * Les paramètres 'from' et 'len' peuvent etre NULL. */
    len = sizeof (from);
    f = accept (ss, (struct sockaddr *)&from, &len);

    /* ... */
}
```

Attention : il manque l'indispensable traitement d'erreur, au moins sur connect, bind, accept.

3.4 Socket en mode datagramme

Attention : il manque l'indispensable traitement d'erreur, au moins sur bind, sendto, recvfrom.

3.4.1 Initialisation côté client

```
struct sockaddr_in adrserv;

struct timeval tv = { AUTH_CLIENT_TIMEOUT, 0 };
int sc, val;
/* Server address. */
bzero (&adrserv, sizeof(adrserv));
adrserv.sin_family = AF_INET;
adrserv.sin_port = htons (serverport);
adrserv.sin_addr.s_addr = htonl(INADDR_BROADCAST);
/* Open a UDP socket */
sc = socket (AF_INET, SOCK_DGRAM, 0);
/* Allow broadcast on this socket. */
val = 1;
setsockopt (sc, SOL_SOCKET, SO_BROADCAST, &val, sizeof(val));
/* Timeout on receive. */
setsockopt (sc, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));

/* and now send a message */
sendto (auth_socket, buf, len, 0, (struct sockaddr *)&adrserv, sizeof(adrserv));
```

3.4.2 Initialisation côté serveur

```
int sserv;
struct sockaddr_in adrserv;
struct sockaddr_in adrcli;
int adrclilen;
/* Open a socket DGRAM (UDP) */
sserv = socket (AF_INET, SOCK_DGRAM, 0);
/* Bind it */
bzero (&adrserv, sizeof(adrserv));
adrserv.sin_family = AF_INET;
adrserv.sin_addr.s_addr = htonl (INADDR_ANY);
adrserv.sin_port = htons (myport);
bind (sserv, (struct sockaddr*)&adrserv, sizeof(adrserv));

/* Now ready to receive. */
bzero (&adrcli, sizeof (adrcli));
adrclilen = sizeof (adrcli);
recvfrom (sserv, buf, len, 0, (struct sockaddr*)&adrcli, &adrclilen);
```

3.4.3 Échanges

```
ssize_t sendto(int sockfd, void *buf, size_t len, int flags,
                    struct sockaddr *dest_addr, socklen_t addrlen);
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                    struct sockaddr *src_addr, socklen_t *addrlen);
```