

# RÉSEAU ET COMMUNICATION

*Notes de Cours/TD/TP autorisées; autres documents, calculatrices, ordinateurs interdits.*

Vous pouvez appeler sans les recopier les fonctions de la « boîte à outil réseau » vues en TD.

## I. Envoi de courrier électronique en TCP/IP

Les courriers électroniques, ou *email*, sont le plus souvent acheminés par le protocole SMTP (Single Mail Transfer Protocol). Ce protocole texte est très simple et sera détaillé dans la suite.

On se propose de réaliser en C un client mail, capable de se connecter au service SMTP d'une machine (port 25 TCP/IP) et de lui transmettre le message dans le protocole SMTP. Le serveur courrier de cette machine traite la connexion, répond par des codes d'erreurs, puis achemine le courrier : si le destinataire est en local, il écrit le email dans sa boîte de messagerie, sinon il contacte le serveur SMTP à l'adresse du destinataire pour lui transmettre le message.

1) Écrire la fonction `int connecter_au_serveur (char *serveur, int port)` recevant en paramètres le nom d'un `serveur` et un numéro de `port`. La fonction crée une socket TCP/IP, se connecte au `port` du `serveur` puis renvoie la socket connectée. Dans tous les cas d'erreur, la fonction affiche le message d'erreur approprié, ferme la socket si besoin et renvoie -1.

2) Écrire la fonction `int envoyer_buf (int soc, char *buf, int len)` recevant en paramètres une socket connectée `soc`, un buffer `buf` et une longueur `len`. La fonction écrit les `len` premiers caractères de `buf` dans la socket (avec autant d'opérations d'écriture que nécessaire). Elle renvoie 0 pour succès, -1 si erreur.

3) Le protocole SMTP est un protocole texte, que l'on peut donc utiliser avec un simple `telnet`. Voici un exemple complet :

<code>\$ telnet adr-serveur 25</code>	<code>Connected to adr-serveur</code> <code>220 adr-serveur SMTP Ready</code>
<code>HELO adr-client</code>	<code>250 adr-serveur</code>
<code>MAIL FROM: email-auteur</code>	<code>250 Sender ok</code>
<code>RCPT TO: email-dest</code>	<code>250 Recipient ok</code>
<code>DATA</code>	<code>354 Enter mail, end with "." on a line by itself</code>
<code>Subject: sujet</code> <code>Ceci est le corps</code> <code>du message à envoyer.</code> <code>.</code>	
<code>QUIT</code>	<code>250 Ok</code>
<code>\$</code>	<code>221 Closing connection</code> <code>Connection closed by foreign host.</code>

Dans cet exemple, on se connecte via `telnet` au serveur `adr-serveur` sur le port 25, on se présente comme le client `adr-client`, puis on décrit le courrier en donnant les informations suivantes : l'adresse `email-auteur` de l'auteur, l'adresse `email-dest` du destinataire, le `sujet` et enfin le corps du message. Dans la colonne de gauche sont données les commandes tapées (`$` désigne le prompt du shell), dans la colonne de droite figure l'affichage obtenu (dont les réponses du serveur). Chaque ligne est terminée par un retour chariot '`\n`', non représenté ici.

Écrire la fonction `int ecrire_commandes_smtp (int soc, char *nom_client, Email *e)` recevant en paramètres une socket `soc` connectée au serveur SMTP, le nom du client `nom_client`, et le courrier `e` à envoyer. Ce courrier est décrit par le type suivant :

```
typedef struct {
    char *auteur, *destinataire, *sujet, *corps;
} Email;
```

On supposera pour simplifier que les trois premières chaînes ne contiennent aucun retour chariot, mais on ne préjugera pas de la taille des chaînes.

La fonction écrit toutes les commandes SMTP et informations nécessaires pour transmettre le courrier `e` au serveur (sans se préoccuper des réponses) à l'aide de `envoyer_buf`. À la moindre erreur, la fonction s'interrompt et renvoie -1. Un retour 0 signifie que la fonction a réussi.

4) Après la plupart des commandes, le serveur répond une ou plusieurs lignes, chacune terminée par un retour chariot '`\n`'. Chacune de ces lignes commence par un code à trois chiffres puis un message explicatif (les serveurs HTTP font de même). Le premier chiffre est le plus important : 2 signifie que la demande a été exécutée sans erreur, 3 que la demande est en cours d'exécution, 4 indique une erreur temporaire, 5 que la demande n'est pas valide et n'a pas pu être traitée. Autrement dit, 2 et 3 valent succès, 4 et 5 erreur.

Écrire la fonction `int tester_erreur_smtp (char *ligne)` recevant en paramètre une ligne terminée par un retour chariot. La fonction examine le premier chiffre du code, puis renvoie 0 si le code est un succès, -1 si le code est un échec. Dans ce dernier cas, la fonction affiche un message d'erreur avec la ligne incriminée.

5) Écrire la fonction `int lire_reponses_smtp (int soc)` recevant en paramètre une socket connectée au serveur SMTP. La fonction lit toutes les réponses du serveur jusqu'à la déconnexion, et pour chaque ligne détectée, teste les réponses SMTP. La fonction s'interrompt à la première erreur détectée et affiche la ligne déclarant l'erreur. La fonction renvoie 0 pour succès, -1 pour erreur.

6) Écrire la fonction `int envoyer_email (char *nom_serveur, int port, char *nom_client, Email *e)` recevant en paramètres le nom `nom_serveur` d'un serveur SMTP, le `port` du service SMTP, le nom `nom_client` du client et le courrier `e`. La fonction se connecte au serveur, écrit les commandes SMTP pour envoyer `e`, puis lit les réponses SMTP et enfin ferme la connexion. À la moindre erreur, la fonction s'interrompt et renvoie -1. Elle renvoie 0 si tout s'est bien passé.

7) Écrire le programme principal qui attend en arguments le serveur, le port, le client, l'auteur, le destinataire, le sujet et le corps du message. Il envoie le courrier puis affiche un message de succès ou d'échec, enfin se termine avec le code de sortie correspondant.

# Correction

Les fonctions de la « boîte à outil réseau » vues en TD peuvent être appelées sans les recopier.

## I. Envoi de courrier électronique en TCP/IP

Les courriers électroniques, ou *email*, sont le plus souvent acheminés par le protocole SMTP (Single Mail Transfer Protocol). Ce protocole texte est très simple et sera détaillé dans la suite.

On se propose de réaliser en C un client mail, capable de se connecter au service SMTP d'une machine (port 25 TCP/IP) et de lui transmettre le message dans le protocole SMTP. Le serveur courrier de cette machine traite la connexion, répond par des codes d'erreurs, puis achemine le courrier : si le destinataire est en local, il écrit le email dans sa boîte de messagerie, sinon il contacte le serveur SMTP à l'adresse du destinataire pour lui transmettre le message.

1) Écrire la fonction `int connecter_au_serveur (char *serveur, int port)` recevant en paramètres le nom d'un `serveur` et un numéro de `port`. La fonction crée une socket TCP/IP, se connecte au `port` du `serveur` puis renvoie la socket connectée. Dans tous les cas d'erreur, la fonction affiche le message d'erreur approprié, ferme la socket si besoin et renvoie -1.

```
int connecter_au_serveur (char *serveur, int port)
{
    struct sockaddr_in adr_cli, adr_ser;
    struct hostent *hp;
    int soc;

    /* Création d'une socket domaine internet et mode connecté */
    soc = socket (AF_INET, SOCK_STREAM, 0);
    if (soc < 0) { perror ("socket ip"); return -1; }

    /* Fabrication adresse du client */
    adr_cli.sin_family = AF_INET;
    adr_cli.sin_port = htons (0); /* 0 pour attribution d'un port libre */
    adr_cli.sin_addr.s_addr = htonl(INADDR_ANY); /* Toutes les adr. locales */

    /* Attachement socket à l'adresse du client */
    printf ("Attachement socket\n");
    if (bor_bind_in (soc, &adr_cli) == -1)
        { perror ("bind ip"); close (soc); return -1; }

    /* Fabrication adresse du serveur */
    adr_ser.sin_family = AF_INET;
    adr_ser.sin_port = htons (port); /* forme Network */
    printf ("Résolution adr serveur ... \n");
    if ((hp = gethostbyname (serveur)) == NULL) /* h_errno, perror() */
        { perror ("gethostbyname ip"); close (soc); return -1; }
    memcpy (&adr_ser.sin_addr.s_addr, hp->h_addr, hp->h_length);

    printf ("Connexion à %s ... \n", serveur);
    if (bor_connect_in (soc, &adr_ser) < 0)
        { perror ("connect"); close (soc); return -1; }
    printf ("Connexion établie\n");

    return soc;
}
```

2) Écrire la fonction `int envoyer_buf (int soc, char *buf, int len)` recevant en paramètres une socket connectée `soc`, un buffer `buf` et une longueur `len`. La fonction écrit les `len` premiers caractères de `buf` dans la socket (avec autant d'opérations d'écriture que nécessaire). Elle renvoie 0 pour succès, -1 si erreur.

```
int envoyer_buf (int soc, char *buf, int len)
{
    int a = 0, b = len, k;

    printf ("Envoi de \"%s\"\n", buf);
    while (b-a > 0) {
        k = write (soc, buf+a, b-a);
        if (k < 0) { perror ("envoyer_buf: write"); return -1; }
        a += k;
    }
    return 0;
}
```

3) Le protocole SMTP est un protocole texte, que l'on peut donc utiliser avec un simple `telnet`. Voici un exemple complet :

\$ telnet <i>adr-serveur</i> 25	Connected to <i>adr-serveur</i> 220 <i>adr-serveur</i> SMTP Ready
HELO <i>adr-client</i>	250 <i>adr-serveur</i>
MAIL FROM: <i>email-auteur</i>	250 Sender ok
RCPT TO: <i>email-dest</i>	250 Recipient ok
DATA	354 Enter mail, end with "." on a line by itself
Subject: <i>sujet</i> <i>Ceci est le corps</i> <i>du message à envoyer.</i> .	250 Ok
QUIT	221 Closing connection Connection closed by foreign host.
\$	

Dans cet exemple, on se connecte via `telnet` au serveur *adr-serveur* sur le port 25, on se présente comme le client *adr-client*, puis on décrit le courrier en donnant les informations suivantes : l'adresse *email-auteur* de l'auteur, l'adresse *email-dest* du destinataire, le *sujet* et enfin le corps du message. Dans la colonne de gauche sont données les commandes tapées (\$ désigne le prompt du shell), dans la colonne de droite figure l'affichage obtenu (dont les réponses du serveur). Chaque ligne est terminée par un retour chariot '\n', non représenté ici.

Écrire la fonction `int ecrire_commandes_smtp (int soc, char *nom_client, Email *e)` recevant en paramètres une socket `soc` connectée au serveur SMTP, le nom du client `nom_client`, et le courrier `e` à envoyer. Ce courrier est décrit par le type suivant :

```
typedef struct {
    char *auteur, *destinataire, *sujet, *corps;
} Email;
```

On supposera pour simplifier que les trois premières chaînes ne contiennent aucun retour chariot, mais on ne préjugera pas de la taille des chaînes.

La fonction écrit toutes les commandes SMTP et informations nécessaires pour transmettre le courrier e au serveur (sans se préoccuper des réponses) à l'aide de `envoyer_buf`. À la moindre erreur, la fonction s'interrompt et renvoie -1. Un retour 0 signifie que la fonction a réussi.

```
int ecrire_commandes_smtp (int soc, char *nom_client, Email *e)
{
    char *buf_v[100];
    int buf_n = 0, i;

    buf_v[buf_n++] = "HELO ";
    buf_v[buf_n++] = nom_client;
    buf_v[buf_n++] = "\nMAIL FROM: ";
    buf_v[buf_n++] = e->auteur;
    buf_v[buf_n++] = "\nRCPT TO: ";
    buf_v[buf_n++] = e->destinataire;
    buf_v[buf_n++] = "\nDATA\nSubject: ";
    buf_v[buf_n++] = e->sujet;
    buf_v[buf_n++] = "\n";
    buf_v[buf_n++] = e->corps;
    buf_v[buf_n++] = "\n.\nQUIT\n";

    for (i = 0; i < buf_n ; i++) {
        if (envoyer_buf (soc, buf_v[i], strlen(buf_v[i])) < 0)
            return -1;
    }

    return 0;
}
```

4) Après la plupart des commandes, le serveur répond une ou plusieurs lignes, chacune terminée par un retour chariot '\n'. Chacune de ces lignes commence par un code à trois chiffres puis un message explicatif (les serveurs HTTP font de même). Le premier chiffre est le plus important : 2 signifie que la demande a été exécutée sans erreur, 3 que la demande est en cours d'exécution, 4 indique une erreur temporaire, 5 que la demande n'est pas valide et n'a pas pu être traitée. Autrement dit, 2 et 3 valent succès, 4 et 5 erreur.

Écrire la fonction `int tester_erreur_smtp (char *ligne)` recevant en paramètre une ligne terminée par un retour chariot. La fonction examine le premier chiffre du code, puis renvoie 0 si le code est un succès, -1 si le code est un échec. Dans ce dernier cas, la fonction affiche un message d'erreur avec la ligne incriminée.

```
int tester_erreur_smtp (char *ligne)
{
    if (ligne[0] == '2' || ligne[0] == '3')
        return 0;
    if (ligne[0] == '4' || ligne[0] == '5') {
        fprintf (stderr, "ERREUR SMTP: %s\n", ligne); return -1;
    }
    fprintf (stderr, "AUTRE ERREUR: %s\n", ligne);
    return -1;
}
```

5) Écrire la fonction `int lire_reponses_smtp (int soc)` recevant en paramètre une socket connectée au serveur SMTP. La fonction lit toutes les réponses du serveur jusqu'à la déconnexion, et pour chaque ligne détectée, teste les réponses SMTP. La fonction s'interrompt à la première erreur détectée et affiche la ligne déclarant l'erreur. La fonction renvoie 0 pour succès, -1 pour erreur.

```
int lire_reponses_smtp (int soc) /* voir TD7 "défragmentation de lignes" */
{
    char buf[4096];
    int i, k, pos = 0, buf_size = sizeof(buf);

    while (1)
    {
        k = read (soc, buf+pos, buf_size-pos-1);

        /* Traitement des erreurs : si k <= 0, on n'a pas rajouté de
           caractères dans buf, donc buf[0 .. pos-1] ne contient aucun '\n' */
        if (k < 0) { perror ("read socket"); return -1; }
        if (k == 0) return 0;

        buf[pos+k] = 0; /* Par sécurité */

        /* Recherche des lignes : on sait qu'il n'y a pas de '\n' dans
           buf[0 .. pos-1], donc on cherche les '\n' dans buf[pos .. pos+k-1].

           < pas de \n >< partie à analyser >
           buf : [ _ | _ _ _ _ _ _ _ _ | _ | _ _ _ _ | \n | _ _ _ _ _ _ _ _ | 0 | _ _ _ _ _ _ _ _ ]
                  0                pos          i                pos+k
           < supprimé                >< conservé >
        */
        for (i = pos ; i < pos+k; i++)
            if (buf[i] == '\n') {
                buf[i] = 0;

                /* On traite la ligne */
                if (tester_erreur_smtp (buf) < 0) return -1;

                k = pos+k-(i+1); /* nb de car restant à analyser */
                memmove (buf, buf+i+1, k+1); /* copie avec \0 */
                pos = 0; i = -1; /* revient au début de buf */
            }

        pos += k; /* RQ maintenant pos == strlen(buf) */
        if (pos >= buf_size-2) {
            printf ("Dépassement de buf, contenu supprimé\n");
            pos = 0;
        }
    }

    return 0;
}
```

6) Écrire la fonction `int envoyer_email (char *nom_serveur, int port, char *nom_client, Email *e)` recevant en paramètres le nom `nom_serveur` d'un serveur SMTP, le `port` du service SMTP, le nom `nom_client` du client et le courrier `e`. La fonction se connecte au serveur, écrit les commandes SMTP pour envoyer `e`, puis lit les réponses SMTP et enfin ferme la connexion. À la moindre erreur, la fonction s'interrompt et renvoie -1. Elle renvoie 0 si tout s'est bien passé.

```
int envoyer_email (char *nom_serveur, int port, char *nom_client, Email *e)
{
    int soc;

    soc = connecter_au_serveur (nom_serveur, port);
    if (soc < 0) return -1;

    if (ecrire_commandes_smtp (soc, nom_client, e) < 0) {
        close (soc); return -1;
    }
    if (lire_reponses_smtp (soc) < 0) {
        close (soc); return -1;
    }

    close (soc);
    return 0;
}
```

7) Écrire le programme principal qui attend en arguments le serveur, le port, le client, l'auteur, le destinataire, le sujet et le corps du message. Il envoie le courrier puis affiche un message de succès ou d'échec, enfin se termine avec le code de sortie correspondant.

```
int main (int argc, char *argv[])
{
    Email e;
    int port, k;
    char *serveur, *client;

    if (argc != 8) {
        fprintf (stderr, "USAGE: %s serveur port client auteur dest sujet corps\n",
                argv[0]);
        exit (1);
    }
    serveur = argv[1];
    port = atoi (argv[2]);
    client = argv[3];
    e.auteur = argv[4];
    e.destinataire = argv[5];
    e.sujet = argv[6];
    e.corps = argv[7];

    k = envoyer_email (serveur, port, client, &e);

    printf ("%s\n", k == 0 ? "SUCCES" : "ECHEC");
    exit (k == 0 ? 0 : 1);
}
```