Réseau et Communication

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

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

I. Estimation d'heure réseau en UDP/IP

Les ordinateurs synchronisent leur horloge par le réseau avec le protocole NTP (Network Time Protocol). On se propose de réaliser en C un client/serveur de synchronisation d'horloge en UDP/IP, avec un protocole simplifié librement inspiré de NTP.

Le problème à régler est le suivant : lorsqu'un client demande l'heure à un serveur de référence, sa réponse met un certain temps à parvenir au client. Cette durée varie fortement selon le trafic réseau et fait perdre beaucoup en précision. L'idée que nous allons exploiter est d'estimer le temps de réponse à partir de la durée de l'aller-retour, de demander l'heure plusieurs fois et de conserver l'écart (par rapport à l'heure locale) qui a la plus grande précision.

Notre programme peut être à la fois client et serveur. S'il est client, il consulte régulièrement 1 ou plusieurs serveurs de temps de référence, puis en déduit l'écart entre les horloges. Un serveur peut être une racine (son horloge est ultra-précise) ou un relais (il se met à l'heure par le réseau).

On se donne le type `Ref_temps` pour mémoriser l'adresse d'un serveur, et le type `Info` pour mémoriser les paramètres du programme :

```
#define REF_MAX 10                  /* Nb max de serveurs */
typedef struct {
    char *adr_nom; int port;        /* Nom et port du serveur */
    struct sockaddr_in adr;         /* Adresse IPv4 du serveur */
} Ref_temps;

typedef struct {
    Ref_temps ref_liste[REF_MAX];   /* Liste des serveurs */
    int ref_nb;                     /* Taille de la liste */
    struct sockaddr_in adr_locale;  /* Adresse locale */
    int soc, port_local;            /* socket, port local */
    int rappel_ms;                  /* Délai entre 2 consultations, en ms */
    double delta_min, ecart_opt;    /* Précision, écart optimal en s */
    int handle;                     /* Pour le timeout */
} Info;
```

1) Écrire la fonction `void init_info (Info *info)` qui initialise `info` avec une liste vide, le port local 12300, un délai de 2000 ms, la socket, le delta et le handle à -1 et l'écart optimal à 0.

2) On suppose disposer de la fonction `int fabriquer_adr_serveur_ip (int port, char *nom, struct sockaddr_in *adr)` qui résout le `nom` donné en paramètre, puis mémorise l'adresse IPv4 correspondante et le numéro de `port` dans `adr`. Elle renvoie 0 pour succès, -1 pour erreur.

Écrire la fonction `int ajouter_ref (Info *info, char *adr_nom, int port)` qui mémorise le nom et le port dans la liste des serveurs, ainsi que son adresse IPv4. La fonction renvoie 0 pour succès, -1 pour erreur.

3) On se donne la fonction `temps_local` qui renvoie l'heure locale sous la forme d'un double :

```
double temps_local () { struct timeval t; gettimeofday (&t, NULL);
                        return t.tv_sec + t.tv_usec / 1000000.0; }
```

Écrire la fonction `int questionner_les_serveurs (Info *info)`, qui pour chaque serveur de la liste, calcule l'heure locale t_1 puis envoie le message `"LOCT` t_1`"` (avec 6 chiffres décimaux pour avoir une précision en micro secondes). La fonction renvoie 0 pour succès, -1 pour erreur.

4) On se donne la fonction `temps_actuel_corrige` qui renvoie le temps local corrigé, c'est-à-dire le temps local moins l'écart optimal, sous la forme d'un double (si le serveur est racine, alors l'écart optimal est nul) :

```
double temps_actuel_corrige (Info *info) { return temps_local () - info->ecart_opt; }
```

Écrire la fonction `int traiter_LOCT (Info *info, struct sockaddr_in *adr, char *args)` qui traite une requête `"LOCT` t_1`"` et répond au client `adr`. La chaîne de caractères `args` est la suite du mot `LOCT` dans la requête, autrement dit elle doit contenir t_1 (l'heure locale du client).

La réponse sera de la forme `"REFT` t_1 t'_1`"`, où t'_1 est le temps actuel corrigé du serveur (avec 6 chiffres décimaux). Ainsi, le client n'aura pas besoin de mémoriser sa requête pour faire son calcul puisque le serveur lui redonnera t_1 ; de plus, si la requête ou la réponse est perdue, cela n'aura aucun effet. La fonction renvoie 0 pour succès, -1 pour erreur.

5) Écrire la fonction `int traiter_REFT (Info *info, struct sockaddr_in *adr, char *args)` qui traite une réponse `"REFT` t_1 t'_1`"`, calcule l'heure de référence estimée puis l'affiche (en secondes et micro-secondes sans faire de conversion en date). La chaîne de caractères `args` est la suite du mot `REFT` dans la réponse, autrement dit elle doit contenir t_1 et t'_1 (l'heure locale à l'envoi du client et l'heure du serveur). La fonction renvoie 0 pour succès, -1 pour erreur.

La méthode de calcul est la suivante : au début de la fonction, on mémorise l'heure locale t_2 de réception du message. La durée de l'aller-retour entre le client et le serveur est donc $\delta = t_2 - t_1$. Le temps médian local est $m = (t_1 + t_2)/2$. Si l'on suppose que l'aller-retour est symétrique, c'est-à-dire que la durée de l'aller est sensiblement égale à celle de retour, alors le serveur a répondu presque en même temps (t'_1) que l'instant médian. Le client en déduit l'écart $\theta = m - t'_1$ entre les horloges ; cela entraine que au temps local t_2, l'heure de référence estimée est $t_2 - \theta$.

On n'a aucun moyen de savoir si la durée de l'aller-retour est symétrique, donc la précision est directement liée à δ. C'est pourquoi on conserve l'écart optimal dans `info`, qui est l'écart θ pour lequel le δ est minimum. La fonction affiche chaque fois toutes ces valeurs.

6) Écrire la fonction `int faire_un_dialogue (Info *info)` qui attend un datagramme, puis en fonction de l'action (le premier mot du datagramme), appelle `traiter_LOCT` ou `traiter_REFT`. La fonction renvoie 0 pour succès, -1 pour erreur.

7) Écrire la fonction `int passer_le_temps (Info *info)` qui effectue un dialogue si la socket est éligible, ou alors, si le timer est à échéance, questionne tous les serveurs de la liste, puis réarme le timer pour une durée de `info->rappel_ms` ms. La fonction renvoie 0 pour succès, -1 pour erreur.

8) On suppose disposer de la fonction `int creer_socket_udpip (int nport, struct sockaddr_in *adr)` qui crée une socket UDP/IP, l'attache au port `nport` (0 pour attribution) puis affiche le port ouvert. Elle renvoie la socket, ou -1 si erreur.

Écrire la fonction `main` du programme, acceptant les arguments suivants :
`[-p port_local] [-s delai_en_ms] [-ref adr port] [-ref adr port] ...`

La fonction décode les arguments, sinon affiche l'usage et échoue. Elle crée la socket UDP/IP, arme éventuellement le timer si la liste des serveurs de référence n'est pas vide (auquel cas c'est un serveur racine) puis rentre dans une boucle interruptible où elle passe le temps.

Correction

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

I. Estimation d'heure réseau en UDP/IP

Les ordinateurs synchronisent leur horloge par le réseau avec le protocole NTP (Network Time Protocol). On se propose de réaliser en C un client/serveur de synchronisation d'horloge en UDP/IP, avec un protocole simplifié librement inspiré de NTP.

Le problème à régler est le suivant : lorsqu'un client demande l'heure à un serveur de référence, sa réponse met un certain temps à parvenir au client. Cette durée varie fortement selon le trafic réseau et fait perdre beaucoup en précision.

L'idée que nous allons exploiter est d'estimer le temps de réponse à partir de la durée de l'aller-retour, de demander l'heure plusieurs fois et de conserver l'écart (par rapport à l'heure locale) qui a la plus grande précision.

Notre programme peut être à la fois client et serveur. S'il est client, il consulte régulièrement 1 ou plusieurs serveurs de temps de référence, puis en déduit l'écart entre les horloges. Un serveur peut être une racine (son horloge est ultra-précise) ou un relais (il se met à l'heure par le réseau).

On se donne le type `Ref_temps` pour mémoriser l'adresse d'un serveur, et le type `Info` pour mémoriser les paramètres du programme :

```
#define REF_MAX 10                  /* Nb max de serveurs */

typedef struct {
    char *adr_nom;                  /* Nom du serveur */
    struct sockaddr_in adr;         /* Adresse IPv4 du serveur */
    int port;                       /* Numéro de port du serveur */
} Ref_temps;

typedef struct {
    Ref_temps ref_liste[REF_MAX];   /* Liste des serveurs */
    int ref_nb;                     /* Taille de la liste */
    struct sockaddr_in adr_locale;  /* Adresse locale */
    int port_local;                 /* Port local */
    int rappel_ms;                  /* Délai entre 2 consultations, en ms */
    int soc;                        /* Socket UDP/IP */
    double delta_min;               /* Précision en s */
    double ecart_opt;               /* Écart optimal en s */
    int handle;                     /* Pour le timeout */
} Info;
```

1) Écrire la fonction `void init_info (Info *info)` qui initialise `info` avec une liste vide, le port local 12300, un délai de 2000 ms, la socket, le delta et le handle à -1 et l'écart optimal à 0.

```
void init_info (Info *info)
{
    info->ref_nb = 0;
    info->port_local = 12300;
    info->rappel_ms = 2000;
    info->soc = -1;
    info->delta_min = -1;
    info->ecart_opt = 0;
    info->handle = -1;
}
```

2) On suppose disposer de la fonction `int fabriquer_adr_serveur_ip (int port, char *nom, struct sockaddr_in *adr)` qui résout le `nom` donné en paramètre, puis mémorise l'adresse IPv4 correspondante et le numéro de `port` dans `adr`. Elle renvoie 0 pour succès, -1 pour erreur.

Écrire la fonction `int ajouter_ref (Info *info, char *adr_nom, int port)` qui mémorise le nom et le port dans la liste des serveurs, ainsi que son adresse IPv4. La fonction renvoie 0 pour succès, -1 pour erreur.

```
int ajouter_ref (Info *info, char *adr_nom, int port)
{
    Ref_temps *r;

    if (info->ref_nb >= REF_MAX) {
        fprintf (stderr, "ERREUR : trop de serveurs de référence.");
        return -1;
    }
    r = &info->ref_liste[info->ref_nb++];

    r->adr_nom = adr_nom;
    r->port    = port;

    return fabriquer_adr_serveur_ip (r->port, r->adr_nom, &r->adr);
}
```

3) On se donne la fonction `temps_local` qui renvoie l'heure locale sous la forme d'un double :

```
double temps_local ()
{
    struct timeval t;
    gettimeofday (&t, NULL);
    return t.tv_sec + t.tv_usec / 1000000.0;
}
```

Écrire la fonction `int questionner_les_serveurs (Info *info)`, qui pour chaque serveur de la liste, calcule l'heure locale t_1 puis envoie le message `"LOCT` t_1`"` (avec 6 chiffres décimaux pour avoir une précision en micro secondes). La fonction renvoie 0 pour succès, -1 pour erreur.

```
int questionner_les_serveurs (Info *info)
{
    int i, k;
    char buf[1500];
    double t1;

    for (i = 0; i < info->ref_nb; i++) {

        t1 = temps_local ();  /* Dans la boucle, pour conserver précision */
        sprintf (buf, "LOCT %.6f", t1);
        k = bor_sendto_in (info->soc, buf, strlen(buf), &info->ref_liste[i].adr);
        if (k < 0) return -1;
        printf ("Envoyé à %s:%d : %s\n",
            info->ref_liste[i].adr_nom, info->ref_liste[i].port, buf);
    }
    return 0;
}
```

4) On se donne la fonction `temps_actuel_corrige` qui renvoie le temps local corrigé, c'est-à-dire le temps local moins l'écart optimal, sous la forme d'un double (si le serveur est racine, alors l'écart optimal est nul).

```
double temps_actuel_corrige (Info *info)
{
    return temps_local () - info->ecart_opt;
}
```

Écrire la fonction `int traiter_LOCT (Info *info, struct sockaddr_in *adr, char *args)` qui traite une requête "`LOCT` t_1" et répond au client `adr`. La chaîne de caractères `args` est la suite du mot `LOCT` dans la requête, autrement dit elle doit contenir t_1 (l'heure locale du client).

La réponse sera de la forme "`REFT` t_1 t'_1", où t'_1 est le temps actuel corrigé du serveur (avec 6 chiffres décimaux). Ainsi, le client n'aura pas besoin de mémoriser sa requête pour faire son calcul puisque le serveur lui redonnera t_1 ; de plus, si la requête ou la réponse est perdue, cela n'aura aucun effet. La fonction renvoie 0 pour succès, -1 pour erreur.

```
/* Traite la requete du client au format "LOCT T1"
 * On répond "REFT T1 T'1"
*/
int traiter_LOCT (Info *info, struct sockaddr_in *adr, char *args)
{
    char buf[1500];
    double t1, tp1;
    int k;

    if (sscanf (args, "%lf", &t1) != 1) {  /* renvoie 0 pour ne pas stopper */
        fprintf (stderr, "Erreur, double attendu\n"); return 0;
    }
    tp1 = temps_actuel_corrige (info);
    sprintf (buf, "REFT %.06f %.06f", t1, tp1);
    k = bor_sendto_in (info->soc, buf, strlen(buf), adr);
    if (k < 0) return -1;
    printf ("Répondu à %s : %s\n",
        bor_adrtoa_in (adr), buf);

    return 0;
}
```

5) Écrire la fonction `int traiter_REFT (Info *info, struct sockaddr_in *adr, char *args)` qui traite une réponse "`REFT` t_1 t'_1", calcule l'heure de référence estimée puis l'affiche (en secondes et micro-secondes sans faire de conversion en date). La chaîne de caractères `args` est la suite du mot `REFT` dans la réponse, autrement dit elle doit contenir t_1 et t'_1 (l'heure locale à l'envoi du client et l'heure du serveur). La fonction renvoie 0 pour succès, -1 pour erreur.

La méthode de calcul est la suivante : au début de la fonction, on mémorise l'heure locale t_2 de réception du message. La durée de l'aller-retour entre le client et le serveur est donc $\delta = t_2 - t_1$. Le temps médian local est $m = (t_1 + t_2)/2$. Si l'on suppose que l'aller-retour est symétrique, c'est-à-dire que la durée de l'aller est sensiblement égale à celle de retour, alors le serveur a répondu presque en même temps (t'_1) que l'instant médian. Le client en déduit l'écart $\theta = m - t'_1$ entre les horloges ; cela entraine que au temps local t_2, l'heure de référence estimée est $t_2 - \theta$.

On n'a aucun moyen de savoir si la durée de l'aller-retour est symétrique, donc la précision est directement liée à δ. C'est pourquoi on conserve l'écart optimal dans `info`, qui est l'écart θ pour lequel le δ est minimum. La fonction affiche chaque fois toutes ces valeurs.

```
/* Traite la reponse du serveur au format "REFT T1 T'1"
 * On calcule le temps et on l'affiche
*/
int traiter_REFT (Info *info, struct sockaddr_in *adr, char *args)
{
    double t1, tp1, t2, delta_ar, median, ecart;
    (void) adr;  /* paramètre inutilisé */

    if (sscanf (args, "%lf %lf", &t1, &tp1) != 2) {
        fprintf (stderr, "Mauvais format : %s\n", args); return -1;
    }

    t2 = temps_local ();
```

```
    delta_ar = t2-t1;
    median = (t1+t2)/2;
    ecart = median - tp1;

    printf ("Temps local        : %.06f s\n", t2);
    printf ("Temps aller-retour : %.06f s\n", delta_ar);
    printf ("Temps médian       : %.06f s\n", median);
    printf ("Ecart estimé       : %.06f s\n", ecart);
    printf ("Temps estimé       : %.06f s\n", t2-ecart);

    if (info->delta_min < 0 || info->delta_min > delta_ar) {
        info->delta_min = delta_ar;
        info->ecart_opt = ecart;
        printf ("Nouveau delta de référence : %.06f s\n", delta_ar);
        printf ("Nouvel écart de référence  : %.06f s\n", ecart);
    }
    printf ("Meilleur t. estimé : %.06f s\n", t2 - info->ecart_opt);

    return 0;
}
```

6) Écrire la fonction `int faire_un_dialogue (Info *info)` qui attend un datagramme, puis en fonction de l'action (le premier mot du datagramme), appelle `traiter_LOCT` ou `traiter_REFT`. La fonction renvoie 0 pour succès, -1 pour erreur.

```
int faire_un_dialogue (Info *info)
{
    char requete[1500], action[1500];
    struct sockaddr_in adr;
    int k, pos;

    printf ("Lecture ...\n");
    k = bor_recvfrom_in (info->soc, requete, sizeof(requete)-1, &adr);
    if (k < 0) return -1;
    requete[k] = 0;
    printf ("Reçu : %s\n", requete);

    if (sscanf(requete, "%s%n", action, &pos) != 1) {
        fprintf (stderr, "Erreur, action non trouvée.\n");
    } else if (strcmp (action, "LOCT") == 0) {
        return traiter_LOCT (info, &adr, requete+pos);
    } else if (strcmp (action, "REFT") == 0) {
        return traiter_REFT (info, &adr, requete+pos);
    } else {
        fprintf (stderr, "Erreur, action non valide.\n");
    }

    return 0;
}
```

7) Écrire la fonction `int passer_le_temps (Info *info)` qui effectue un dialogue si la socket est éligible, ou alors, si le timer est à échéance, questionne tous les serveurs de la liste, puis réarme le timer pour une durée de `info->rappel_ms` mili-secondes. La fonction renvoie 0 pour succès, -1 pour erreur.

```
int passer_le_temps (Info *info)
{
    fd_set set_read;
    int res, max_fd;

    FD_ZERO (&set_read);
    FD_SET (info->soc, &set_read);
    max_fd = info->soc;
```

```
    res = select (max_fd+1, &set_read, NULL, NULL, bor_timer_delay());

    if (res < 0) {
        if (errno == EINTR) return 0;
        perror ("select"); return -1;
    }

    if (res == 0) {  /* timeout à échéance */
        int handle = bor_timer_handle();
        bor_timer_remove (handle);
        info->handle = bor_timer_add (info->rappel_ms, NULL);
        return questionner_les_serveurs (info);
    }

    if ( ! FD_ISSET (info->soc, &set_read)) {
        fprintf (stderr, "Erreur interne : la socket devrait être éligible\n");
        return -1;
    }

    return faire_un_dialogue (info);
}
```

8) On suppose disposer de la fonction `int creer_socket_udpip (int nport, struct sockaddr_in *adr)` qui crée une socket UDP/IP, l'attache au port `nport` (0 pour attribution) puis affiche le port ouvert. Elle renvoie la socket, ou -1 si erreur.

Écrire la fonction `main` du programme, acceptant les arguments suivants :
`[-p port_local] [-s delai_en_ms] [-ref adr port] [-ref adr port] ...`

La fonction décode les arguments, sinon affiche l'usage et échoue. Elle crée la socket UDP/IP, arme éventuellement le timer si la liste des serveurs de référence n'est pas vide (auquel cas c'est un serveur racine) puis rentre dans une boucle interruptible où elle passe le temps.

```
int boucle_princ = 1;
void capter_sig (int sig)
{
    printf ("Signal %d capté\n", sig);
    boucle_princ = 0;
}

void afficher_usage (FILE *f, char *nom_prog)
{
    fprintf (f, "USAGE: %s [-p port_local] [-s delai_en_ms] [-ref adr port]  ...\n",
             nom_prog);
}

int main (int argc, char *argv[])
{
    Info info;
    char *nom_prog = argv[0];
    int res = 0;

    init_info (&info);

    while (argc > 1) {
        if (argc > 2 && strcmp(argv[1], "-p") == 0) {
            info.port_local = atoi(argv[2]);
            argc -= 2; argv += 2;
        } else if (argc > 2 && strcmp(argv[1], "-s") == 0) {
            info.rappel_ms = atoi(argv[2]);
            argc -= 2; argv += 2;
        } else if (argc > 3 && strcmp(argv[1], "-ref") == 0) {
            if (ajouter_ref (&info, argv[2], atoi(argv[3])) < 0) exit(1);
            argc -= 3; argv += 3;
```

```
        } else {
            afficher_usage (stderr, nom_prog); exit (1);
        }
    }

    info.soc = creer_socket_udpip (info.port_local, &info.adr_locale);
    if (info.soc < 0) exit (1);

    if (info.ref_nb > 0)
        info.handle = bor_timer_add (info.rappel_ms, NULL);

    bor_signal (SIGINT, capter_sig, 0);

    while (boucle_princ)
        if ((res == passer_le_temps (&info)) < 0) break;

    close (info.soc);
    exit (res < 0 ? 1 : 0);
}
```

Pour être complet, voici le code des fonctions supposées données.

```
int creer_socket_udpip (int nport, struct sockaddr_in *adr)
{
    int soc;

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

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

    /* Attachement socket à l'adresse locale */
    if (bor_bind_in (soc, adr) == -1)
    { close (soc); return -1; }

    /* Récupération de l'adresse réelle et du port sous forme Network */
    if (bor_getsockname_in (soc, adr) < 0)
      { close (soc); exit(1); }
    printf ("port %d ouvert\n", ntohs(adr->sin_port));

    return soc;
}

int fabriquer_adr_serveur_ip (int nport, char *nom, struct sockaddr_in *adr)
{
    struct hostent *hp;

    adr->sin_family = AF_INET;
    adr->sin_port = htons (nport); /* forme Network */
    printf ("Résolution adr %s ...\n", nom);
    if ((hp = gethostbyname (nom)) == NULL)   /* h_errno, herror() */
      { herror ("gethostbyname"); return -1; }
    memcpy (&adr->sin_addr.s_addr, hp->h_addr, hp->h_length);

    return 0;
}
```