IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

TP - Analyse des protocoles IP, TCP et UDP

UE PRIP

Automne 2023

Vous trouverez en annexe une feuille qui vous permettra de noter exactement quelles commandes et quels paramètres vous utiliserez pour configurer le réseau ainsi que les réponses aux questions posées dans ce TP.

1 Mise en place du réseau et paramétrages

Le réseau de test est architecturé comme le montre la figure 1.

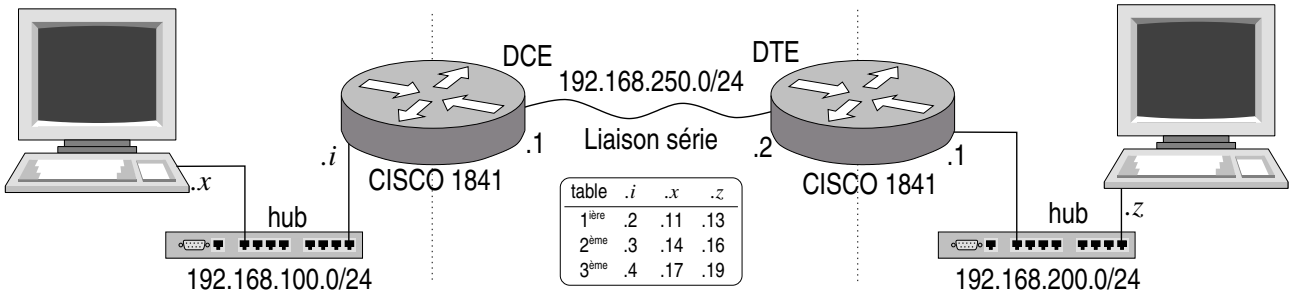


FIGURE 1 – Réseau de test

Les machines terminales sont des PCs munis du système d'exploitation Linux. Le mot de passe administrateur vous sera donné en début de séance. La configuration a déjà été réalisée.

Les routeurs CISCO sont déjà préconfigurés (nous apprendrons à les configurer dans un TP ultérieur). Entre ces routeurs a été mis en place une liaison série (encapsulation HDLC, débit 64Kb/s).

Dans cette partie du TP il vous est demandé de relever la configuration réseau de vos machines : notez leurs adresses IP (IPv4 et IPv6 <scope global>), ainsi que leur route par défaut.

Rappels sur les commandes Unix réseau :

ip address (que l'on peut raccourcir en `ip addr`) permet d'afficher ou attribuer une addresse IP à une interface internet. [1] Exemples :
`man ip` et `man ip-address` : la documentation de cette commande, à lire en premier
`ip addr` : liste toutes les interfaces disponibles
`if addr eno2 192.168.xxx.yyy/24 broadcast 192.168.xxx.255 up`

ip route permet de gérer la table de routage de la machine. [2] Exemple :
`man ip` et `man-route` : la documentation de cette commande, à lire en premier
`ip route` : afficher la table de routage
`ip route add default gw 192.168.xxx.zzz` : positionne la route par défaut, i.e. l'adresse IP du routeur pour 'sortir'
`ip route del ...` : pour détruire une route.

ping (ping-pong) permet de tester la liaison IP vers une autre machine. Concrètement des paquets ICMP ECHO font l'aller-retour vers l'addresse IP testée. On vérifie ainsi que la configuration IP (adresses, routes, etc.) fonctionne correctement. Exemple :
`ping -n 192.168.100.11` (faire Ctrl-C pour arrêter)

ethtool permet de gérer les fonctions matérielles des cartes ethernet modernes (optionnel).
Exemple : `ethtool eno2` : affiche les possibilités de la carte ethernet.

ip est une commande générique et plus moderne qui tend à remplacer `ifconfig` `route` etc.
Exemple : `ip add show`, ou `ip route show`

1. Les nostalgiques peuvent utiliser la commande `ifconfig` qui fait quasiment la même chose.
2. Les nostalgiques peuvent utiliser la commande `route` qui fait quasiment la même chose.

Pour comprendre un peu mieux comment fonctionne les protocoles IP, TCP, UDP, il faut rendre le système un peu moins intelligent... Les commandes indiquées ci-dessous ont un but purement pédagogique, et en temps normal on laisse généralement les optimisations par défaut. Notez que ces paramétrages ont déjà été fait pour préparer ce TP.

— Si l'*autotuning TCP/IP* est activé (ce qui est souvent le cas par défaut sur Linux), le système s'autorise à changer dynamiquement la taille du buffer de réception (et parfois ré-augmenter la fenêtre TCP). Nous avons donc désactivé cela sur nos PCs pour garder un comportement TCP plus basique [3] :

```
# Disable TCP autotuning
sysctl -w net.ipv4.tcp_moderate_rcvbuf=0
# Force TCP receive buffers to a static size
sysctl -w net.ipv4.tcp_rmem="87380 87380 87380"
# Flush current routing cache
sysctl -w net.ipv4.route.flush=1
```

Pour information, le paramétrage par défaut est :

```
sysctl -w net.ipv4.tcp_moderate_rcvbuf=1
sysctl -w net.ipv4.tcp_rmem="4096 87380 3416064"
```

— L'*offload* sur une carte ethernet consiste à faire réaliser par l'électronique de la carte des opérations qui relèvent plutôt des couches IP-TCP-UDP du système d'exploitation (comme le calcul du checksum des paquets IP, ou le découpage en fonction du MTU des segments TCP). Certes, si la carte sait le faire, cela augmente les performances du système, mais ce mélange des rôles est nuisible sur le plan pédagogique... En fait, l'analyseur de protocole que nous allons utiliser risque de capturer des paquets sortants à une étape dans le système d'exploitation où ils ne sont pas encore complètement bien formés. Nous avons donc demander la désactivation de tout ce qui est possible de désactiver sur notre carte Ethernet avec :

```
ethtool -offload eno2 rx off tx off sg off tso off ufo off gso off gro off lro off
rxhash off
```

On peut obtenir quelques messages d'erreur si certaines fonctions ne peuvent pas être désactivées. C'est sans grande importance. Pour afficher ce qui a pu être désactivé : `ethtool -show-offload eno2`

wireshark Par la suite, sur chacun des PC, nous allons utiliser l'outil `wireshark` (anciennement appelé `ethereal`). Cet outil place la carte ethernet en mode *promiscuous* (cf. `man ifconfig`) : dans ce mode la carte remonte à l'OS tous les paquets ethernet qu'elle voit passer sur le bus réseau ethernet. Ainsi, `wireshark` est capable de capturer tous les paquets qui transitent sur le réseau ethernet auquel est raccordée la machine, et de les afficher de manière sympathique et lisible par un humain.

Toujours dans un but pédagogique, nous avons (déjà) modifié la configuration initiale de la manière suivante :

— Dans le menu `Capture / Options`, cocher `Update list of packet in real time` et `Automatic scrolling in live capture` (on visualise mieux l'aspect dynamique des échanges de paquets), et décocher `Enable network name resolution` (les requêtes DNS ont tendance à polluer la capture).

— Dans `Edit / Preferences / Protocoles / IPv4` décochez :
  — `Reassemble fragmented IP datagrams` (Car justement on veut voir ça dans le TP d'aujourd'hui.)
  — `Validate the IP checksum if possible` (Au cas où la commande `ethtool` de la section récédante n'aurait pas pu désactiver `tx-checksumming`.)
  — `Enable IPv4 geolocaltion` (Ça, c'est franchement inutile.)

— Dans `Edit / Preferences / Protocoles / IPv6` décochez :
  — `Reassemble fragmented IPv6 datagrams`
  — `Enable IPv6 geolocaltion`

— Dans `Edit / Preferences / Protocoles / TCP` décochez :
  — `Validate the TCP checksum if possible`
  — `Relative sequence numbers` (C'est finalement intéressant de voir que ça ne commence pas réellement à 1...)
  — `Analyze TCP sequence numbers` (On va analyser ça nous même dans ce TP.)

— Dans `Edit / Preferences / Protocoles / UDP` décochez :
  — `Validate the UDP checksum if possible`

2 Étude de UDP

Allez dans le répertoire `/opt/TP/TPAnalyseProto`. Vous allez utiliser les programmes `recepteurUDP` et `emetteurUDP` dont vous trouverez les sources en annexe.

3. Cela a déjà été fait par le script `./configure`, inutile de taper ces commandes

— Le recepteur lit des données de taille maximale 100000 octets. On le lancera sur l'un des PC de la manière suivante :
`./recepteurUDP` *`NuméroPort`*

Le numéro de port doit être supérieur à 7000 pour éviter des conflits avec d'autres applications pouvant utiliser des ports inférieurs à cette valeur (par exemple 8003).

— L'émetteur ne peut émettre qu'un seul message de taille paramétrable fournie en argument au lancement. Sur un autre PC, on lancera l'émetteur de la manière suivante :
`./emetteurUDP` *`numéroIPdistant NuméroPortRécepteur nbOctetsMessage`*

Questions :

1. Analysez le code C de l'émetteur et du récepteur. Quelle primitive (fonction) permet d'indiquer l'utilisation du protocole UDP ? Quelle est la taille maximale des messages que l'émetteur est capable de construire ? (Note : lisez le `man` des fonctions et appels systèmes que vous découvrez dans le code C.)
2. Lancez l'exécution du récepteur sur une machine d'extrémité puis lancez l'émetteur sur l'autre machine. Les programmes affichent sur leur console la taille des buffers de socket alloués par l'OS pour la transmission et la réception (buffers alloués dans la couche UDP et non dans les programmes eux-mêmes). Quelle est cette taille ?
3. Quel est le volume de données maximum que peut envoyer réellement l'émetteur ? Essayez en IPv4 et en IPv6 (sur l'adresse de scope global). Comparez ce volume à la réponse à la question 1.
4. Lancez l'émetteur en lui faisant envoyer un message 1000 octets puis un autre de 2000 octets.

   Faites un chronogramme des échanges en étudiant les trames échangées. Éventuellement, vérifiez votre travail en sélectionnant le menu `Statistics / Flow Graphs`.
5. Pour le message de 2000 octets vous constatez une fragmentation. Quelle couche opère cette opération ? Quelle en est la cause ?
6. Comment s'opère cette fragmentation ?
7. Que se passe t'il si le récepteur ne lit pas les données envoyées par l'émetteur ? Si l'émetteur se trompe de numéro de port ? Si l'émetteur se trompe d'adresse IP récepteur (essayez une mauvaise adresse dans le réseau local de l'émetteur, puis une mauvaise adresse un autre réseau local derrière le routeur, et enfin une adresse privée non routable dans Internent et non routée à l'intérieur de l'école p.ex. 172.16.16.16) ?

3 Étude de TCP

Nous allons maintenant utiliser deux programmes communiquant via TCP. Un serveur (de nom `lazyServerTCP`) et un client (`clientTCP`). Vous trouverez en annexe le code source C de ces deux programmes.

Le serveur `lazyServerTCP` accepte les connexions demandées par les clients mais il ne consomme pas les données envoyées par ceux-ci (il ne lit pas les données). On simule ainsi un serveur qui serait ralenti par le traitement au fil de l'eau des données reçues, ce ralentissement serait tel qu'il ne pourrait pas lire le flux entrant pendant un temps relativement long.

Les données reçues remplissent peu à peu le buffer socket de réception et TCP met en œuvre son contrôle de flux afin d'avertir l'émetteur de réduire son taux d'émission, voire de l'arrêter.

Le client est configuré pour émettre sans fin des messages de 100Ko.

— Sur la machine Linux de numéro IP `192.168.200.`*`x`* lancez le serveur en lui fournissant en paramètre un numéro de port supérieur à 7000 (pour éviter des conflits avec d'autres applications pouvant utiliser des ports inférieurs à cette valeur) :
`./lazyServerTCP` *`numéroPort`*

— Lancez l'analyseur `wireshark` et configurez-le comme précédemment.

— Sur la machine Linux de numéro IP `192.168.100.`*`x`* lancez l'analyseur `wireshark` dans les même conditions que sur la machine serveur.

— Sur cette même machine lancez le programme `clientTCP` de la manière suivante :
`./clientTCP` *`numéroIPserveur numéroPort`*

Questions :

1. Analysez le code du client et du serveur. Quelle primitive permet de préciser l'utilisation du protocole TCP ?
2. Faites un chronogramme des échanges des 4 premiers segments TCP en précisant en particulier les numéros de séquence et d'acquittement.

3. Quelles sont les options de TCP utilisées lors de la connexion et lors des échanges ? (l'usage de ces options n'est pas obligatoire, certaines implémentations de TCP ne les mettent pas en œuvre, tout au moins les nouvelles options). Quelle est l'utilité de ces options ?
4. Les programmes client et serveur affichent la taille de leurs buffers de socket, quelle est cette taille ?
5. À la différence d'UDP, TCP met en place un mécanisme de contrôle de flux permettant au récepteur d'asservir le débit de l'émetteur à ses capacités de traitement. Comment s'opère ce contrôle de flux TCP ?
6. Quel est le volume de données émis dans chaque message porteur de données d'application ? Comment expliquez-vous la valeur trouvée ?
7. Quel est le volume de données émis par le client avant que le flux ne soit stoppé par les messages du serveur ? Comment expliquez vous cette valeur (pensez au temps de traversée) ?
8. Du côté client et du côté serveur, relevez les valeurs de `Recv-Q` et `Send-Q` à l'aide de la commande `netstat -tn` (lisez le man). La valeur du `Recv-Q` (côté serveur) doit vous faire penser à quelque chose ; de même la somme de `Recv-Q` (serveur) et `Send-Q` (client).
   Combien d'octets le client dit avoir déjà envoyé avant d'être mis en attente par l'OS sur le `write()` suivant ? [4]
9. Faites un chronogramme des messages échangés, vu du coté client puis vu du coté serveur (faites deux chronogrammes).

4 PATH MTU Discovery

Vous allez étudier maintenant comment s'opère l'adaptation de TCP à la plus petite valeur de MTU (Maximum Transmit Unit) sur le chemin entre l'émetteur et le récepteur (à noter que le client comme le serveur sont à la fois émetteur et récepteur).

Arrêtez le serveur et le client lancés pour le test précédent.

Configurez le MTU de la liaison série entre les routeurs pour qu'il soit de 700. Il faut intervenir sur le routeur qui est du côté de la machine "cliente". Sur le PC qui administre ce routeur, lancez l'émulateur de terminal `gtkterm` et tapez :

```
cisco1#show interfaces Serial 0/0/0 (repérer le MTU actuel)
cisco1#configure terminal
cisco1(config)#interface Serial 0/0/0
cisco1(config-if)#mtu 700
cisco1(config-if)#exit
cisco1(config)#exit
cisco1#show interfaces Serial 0/0/0 (repérer le nouveau MTU)
cisco1#exit
Ctrl-A Q
```

Relancez le programme `lazyServerTCP`, l'analyseur `wireshark` coté client et le programme `clientTCP` dans les mêmes conditions que précédemment.

Questions :

1. Quels sont les messages échangés en début de session et entre quels organes sont ils échangés ? (faites un chronogramme)
2. Comment s'opère l'adaptation au nouveau MTU ? Y a t'il segmentation au niveau IP ?
3. Arrêtez le client et le serveur. Stoppez la capture `wireshark` puis relancez la. Relancez le serveur puis le client. Examinez les traces des échanges, quelles sont les différences ? Quelle conclusion peut-on en tirer ?
4. Comment se passeraient des envois de même volume avec UDP ? (vérifiez au besoin avec les programmes UDP utilisés précédemment). Y a t'il mise en œuvre du *PMTU Discovery* ?
5. Retrouvez ce Path MTU dans le cache de la table de routage avec `ip route show table cache` ou `ip route get <address>`
6. Si vous avez besoin de recommancer vos essais, pensez à purger le cache de la table de routage : `ip route flush cache`.

4. Notez que la relation avec `SO_SNDBUF` et `SO_RCVBUF` est assez compliquée : Linux s'autorisant à faire de l'*autotunning*, ou prend en compte ces paramètres de manière spécifique. Voir le `man 7 tcp`, et `http://www.psc.edu/networking/projects/tcptune/#Linux`

7. Dernier point indépendant des précédents : en conservant la configuration présente et en utilisant les programmes client et serveur UDP, comparez la segmentation IP faite par Linux (voir exercice 2) et par les routeurs Cisco. Vous inhiberez le *PATH MTU Discovery* de Linux par la commande : `sysctl -w net.ipv4.ip_no_pmtu_disc=1`

   Cette commande positionne un paramètre du noyau accessible via le pseudo fichier `/proc/sys/net/ipv4/ip_no_pmtu_disc`.

   Notez qu'il y a d'autres paramètres ajustables concernant le *PATH MTU Discovery* : `/proc/sys/net/ipv4/route/min_pmtu` la taille minimum acceptée (552 par défaut), `/proc/sys/net/ipv4/route/mtu_expires` le temps pendant lequel l'information est gardée en mémoire par le noyau (600s par défaut).

5 Contrôle de congestion

Nous avons vu que dans TCP, le récepteur peut avertir l'émetteur qu'il est congestionné en réduisant la valeur de la *windows size* dans les paquet qu'il lui retourne (la taille de son buffer de réception). C'est la gestion de la congestion de bout en bout, facile.

Plus difficile, la gestion de la congestion sur le chemin consiste pour l'émetteur à estimer le nombre de paquets qu'il peut émettre sur le réseau avant de s'attendre à recevoir un acquittement. Ce nombre de paquets est la fenêtre d'anticipation, appelée encore fenêtre de congestion dans le jargon TCP. Il existe de nombreux algorithmes pour se choisir une bonne fenêtre d'anticipation. [5] Au jour d'aujourd'hui, l'algorithme par défaut sous Linux est *Cubic*.

Nous n'étudierons pas ces algorithmes complexes. Par contre, on peut regarder le résultat. Utilisez la commande `ss` (*Socket Statistics*, lisez le `man`) qui donne des informations un peu plus poussées que `netstat` :

`/sbin/ss -i -e -t -p`

Retrouvez vos sockets TCP. Suivant leur état (connecté, en cours, en attente, etc.) vous aurez diverses informations internes : les options TCP utilisées (timestamp, acquittement sélectif, window scale, etc.), le `rtt` mesuré sur la connexion (moyenne/variance), le `rto` (*Retransmission Time Out*), le `ato` (*delayed Acknowledgment Time Out*), le `ssthresh` (*Slow Start Threshold*), et le fameux `cwnd` (*Congestion Window*).

Notez que la commande `ip route show table cache` vue précédemment vous donne certaines de ces valeurs gardées en cache.

5. `http://en.wikipedia.org/wiki/TCP_congestion_avoidance_algorithm`

Annexe A

Code source C

Listing 1 – emetteurUDP.c

```
/* Auteur: Alain Leroy, Christophe Lohr - Telecom Bretagne */
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define BUFSIZE 100000

int main(int argc, char **argv) {
  int sfd, s, nb, ssz, rsz;
  struct addrinfo hints;
  struct addrinfo *result, *rp;
  struct sockaddr *sa;
  socklen_t salen, lg;
  char buf[BUFSIZE];
  ssize_t nwrite;

  if (argc != 4) {
    printf("Usage : %s nom_machine_recepteur port_recepteur nombre_octets_emettre\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  nb = atoi(argv[3]);
  if (nb <= 0) {
    printf("Donnez un nombre d'octets a emettre superieur a 0\n");
    exit(EXIT_FAILURE);
  }

  /*
   * Obtention de l'adresse IP du distant, a partir de son nom par
   * consultation du fichier /etc/hosts ou de la base hosts des NIS
   * ou du DNS (Domain Name Service)
   * cf. man getaddrinfo(3)
   */
  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = AF_UNSPEC;          /* IPv4 ou IPv6 */
  hints.ai_socktype = SOCK_DGRAM;       /* Datagram socket */
  hints.ai_flags = 0;
  hints.ai_protocol = 0;                /* Any protocol */

  s = getaddrinfo(argv[1], argv[2], &hints, &result);
  if (s != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
    exit(EXIT_FAILURE);
  }

  /* getaddrinfo() retourne une liste de structures d'adresses.
     On essaie chaque adresse jusqu'a ce que socket(2) reussisse. */
  for (rp = result; rp != NULL; rp = rp->ai_next) {
    /* Ouverture de la socket */
    sfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
    if (sfd >= 0)
      break;
  }
  if (rp == NULL) {      /* Aucune adresse valide */
    fprintf(stderr, "Impossible d'ouvrir une socket vers %s\n", argv[1]);
    perror("socket");
    exit(EXIT_FAILURE);
  }

  /*
   * Construction de la structure d'adresse du distant
   */
  sa=malloc(rp->ai_addrlen);
  memcpy(sa, rp->ai_addr, rp->ai_addrlen);
  salen=rp->ai_addrlen;

  freeaddrinfo(result); /* Plus besoin */


  /* Option des sockets: taille des buffers */
  lg = sizeof(rsz);
  if ( getsockopt(sfd, SOL_SOCKET, SO_RCVBUF, &rsz, &lg) == 0 )
    printf("SO_RCVBUF par defaut: %d octets\n", rsz);
  else
    perror("getsockopt SO_RCVBUF");

```

```
    lg = sizeof(ssz);
    if ( getsockopt(sfd, SOL_SOCKET, SO_SNDBUF, &ssz, &lg) == 0 )
      printf("SO_SNDBUF par defaut: %d octets\n", ssz);
    else
      perror("getsockopt SO_SNDBUF");

    rsz = 80000;
    if ( setsockopt(sfd, SOL_SOCKET, SO_RCVBUF, &rsz, sizeof(rsz)) == 0 )
      printf("SO_RCVBUF apres forcage: %d octets\n", rsz);
    else
      perror("setsockopt SO_RCVBUF");


    /* Initialisation du message a transmettre */
    memset(buf, (unsigned char)'a', BUFSIZE);

    /* Ecriture socket */
    nwrite = sendto(sfd, buf, nb, 0, sa, salen);
    printf("Ecrits %zd octets sur la socket\n", nwrite);
    if (nwrite < 0)
      perror("Erreur ecriture");

    close(sfd);

    exit(EXIT_SUCCESS);
  }
```

Listing 2 – recepteurUDP.c

```
  /* Auteur : Alain Leroy, Christophe Lohr - Telecom Bretagne */
  #include <sys/types.h>
  #include <stdio.h>
  #include <stdlib.h>
  #include <unistd.h>
  #include <string.h>
  #include <sys/socket.h>
  #include <netdb.h>

  #define BUFSIZE 100000

  int main(int argc, char **argv) {
    int sfd, s, rsz, ssz;
    struct addrinfo hints;
    struct addrinfo *result, *rp;
    socklen_t lg;
    ssize_t nread;
    char buf[BUFSIZE];
    struct sockaddr_storage peer_addr;
    socklen_t peer_addr_len;
    char host[NI_MAXHOST];


    if (argc != 2) {
      printf("Usage : %s  port_recepteur\n", argv[0]);
      exit(EXIT_FAILURE);
    }

    /* Construction de l'adresse locale (pour bind) */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET6;           /* Force IPv6 */
    hints.ai_socktype = SOCK_DGRAM;       /* Datagram socket */
    hints.ai_flags = AI_PASSIVE;          /* Pour l'adresse IP joker */
    hints.ai_flags |= AI_V4MAPPED|AI_ALL; /* IPv4 remappe en IPv6 */
    hints.ai_protocol = 0;                /* Any protocol */

    s = getaddrinfo(NULL, argv[1], &hints, &result);
    if (s != 0) {
      fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
      exit(EXIT_FAILURE);
    }

    /* getaddrinfo() retourne une liste de structures d'adresses.
       On essaie chaque adresse jusqu'a ce que bind(2) reussisse.
       Si socket(2) (ou bind(2)) echoue, on (ferme la socket et on)
       essaie l'adresse suivante. cf man getaddrinfo(3) */
    for (rp = result; rp != NULL; rp = rp->ai_next) {
      /* Creation de la socket */
      sfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
      if (sfd == -1)
        continue;
      /* Association d'un port a la socket */
      if (bind(sfd, rp->ai_addr, rp->ai_addrlen) == 0)
        break;                  /* Succes */
      close(sfd);
    }
    if (rp == NULL) {      /* Aucune adresse valide */
      perror("bind");
      exit(EXIT_FAILURE);
    }
    freeaddrinfo(result); /* Plus besoin */
```

```
63
    /* Option des sockets: taille des buffers */
    lg = sizeof(rsz);
    if ( getsockopt(sfd, SOL_SOCKET, SO_RCVBUF, &rsz, &lg) == 0 )
      printf("SO_RCVBUF par defaut: %d octets\n", rsz);
68  else
      perror("getsockopt SO_RCVBUF");

    lg = sizeof(ssz);
    if ( getsockopt(sfd, SOL_SOCKET, SO_SNDBUF, &ssz, &lg) == 0 )
73    printf("SO_SNDBUF par defaut: %d octets\n", ssz);
    else
      perror("getsockopt SO_SNDBUF");

    rsz = 80000;
78  if ( setsockopt(sfd, SOL_SOCKET, SO_RCVBUF, &rsz, sizeof(rsz)) == 0 )
      printf("SO_RCVBUF apres forcage: %d octets\n", rsz);
    else
      perror("setsockopt SO_RCVBUF");

83
    /* Boucle de communication */
    for(;;) {
      peer_addr_len = sizeof(struct sockaddr_storage);
      nread = recvfrom(sfd, buf, BUFSIZE, 0,
88                     (struct sockaddr *) &peer_addr, &peer_addr_len);
      if (nread == -1) {
        perror("Erreur en lecture socket\n");
        exit(EXIT_FAILURE);
      } else {
93      s = getnameinfo((struct sockaddr *) &peer_addr,
                        peer_addr_len, host, NI_MAXHOST, NULL, 0, NI_NUMERICHOST);
        if (s == 0)
          printf("Recu %zd octets de %s\n", nread, host);
        else
98        printf("Recu %zd octets (erreur: %s)\n", nread, gai_strerror(s));
      }
    }
  }
```

Listing 3 – clientTCP.c

```
  /* Auteur : Alain Leroy, Christophe Lohr - Telecom Bretagne */
  #include <sys/types.h>
  #include <sys/socket.h>
4 #include <netdb.h>
  #include <stdio.h>
  #include <stdlib.h>
  #include <unistd.h>
  #include <string.h>
9
  #define BUFSIZE 100000

  int main(int argc, char **argv) {
    int sfd, s, ssz, rsz;
14  struct addrinfo hints;
    struct addrinfo *result, *rp;
    socklen_t lg;
    char buf[BUFSIZE];
    ssize_t nwrite;
19

    if (argc != 3) {
      printf("Usage : %s nom_machine_distante port_serveur\n", argv[0]);
      exit(EXIT_FAILURE);
24  }

    /*
     * Obtention de l'adresse IP du distant, a partir de son nom par
     * consultation du fichier /etc/hosts ou de la base hosts des NIS
29   * ou du DNS (Domain Name Service)
     * cf. man getaddrinfo(3)
     */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_UNSPEC;           /* IPv4 ou IPv6 */
34  hints.ai_socktype = SOCK_STREAM;       /* Stream socket */
    hints.ai_flags = 0;
    hints.ai_protocol = 0;                 /* Any protocol */

    s = getaddrinfo(argv[1], argv[2], &hints, &result);
39  if (s != 0) {
      fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
      exit(EXIT_FAILURE);
    }

44
    /* getaddrinfo() retourne une liste de structures d'adresses.
       On essaie chaque adresse jusqu'a ce que connect(2) reussisse.
       Si socket(2) (ou connect(2)) echoue, on (ferme la socket et on)
```

```
      essaie l'adresse suivante. cf man getaddrinfo(3) */
  for (rp = result; rp != NULL; rp = rp->ai_next) {
    /* Ouverture de la socket */
    sfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
    if (sfd == -1)
      continue;
    /* Connexion au distant */
    if (connect(sfd, rp->ai_addr, rp->ai_addrlen) != -1)
      break;              /* Succes */
    close(sfd);
  }
  if (rp == NULL) {      /* Aucune adresse valide */
    perror("connect");
    exit(EXIT_FAILURE);
  }
  freeaddrinfo(result); /* Plus besoin */


  /* Caracteristiques de la socket : taille des buffers */
  /* Taille du buffer de reception */
  lg = sizeof(rsz);
  if ( getsockopt(sfd, SOL_SOCKET, SO_RCVBUF, &rsz, &lg) == 0 )
    printf("SO_RCVBUF: %d octets\n", rsz);
  else
    perror("getsockopt SO_RCVBUF");

  /* Taille du buffer d'emission */
  lg = sizeof(ssz);
  if ( getsockopt(sfd, SOL_SOCKET, SO_SNDBUF, &ssz, &lg) == 0 )
    printf("SO_SNDBUF: %d octets\n", ssz);
  else
    perror("getsockopt SO_SNDBUF");


  /* Initialisation du buffer avec le caractere 'a' */
  memset(buf, (unsigned char)'a', BUFSIZE);

  /* Boucle de communication */
  for(;;) {
    /* Ecriture socket */
    nwrite = write(sfd, buf, BUFSIZE);
    printf("Ecrits %zd octets sur socket\n", nwrite);
  }
}
```

Listing 4 – lazyServerTCP.c

```
/* Auteur : Alain Leroy, Christophe Lohr - Telecom Bretagne */
/*
 * Un serveur sur TCP qui ne fait aucun traitement lors d'une
 * requete d'un client
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <string.h>

//#define BUFSIZE 100000

int main(int argc, char **argv) {
  int sfd, ns, s, rsz, ssz;
  struct addrinfo hints;
  struct addrinfo *result, *rp;
  socklen_t lg;
  //ssize_t nread;
  //char buf[BUFSIZE];
  struct sockaddr_storage from;
  socklen_t fromlen;
  char host[NI_MAXHOST];


  /* Le serveur a besoin de connaitre
   * le numero du port sur lequel il attend les requetes du client
   */
  if (argc != 2) {
    printf("Usage : %s  port_serveur\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  /* Construction de l'adresse locale (pour bind) */
  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = AF_INET6;           /* Force IPv6 */
  hints.ai_socktype = SOCK_STREAM;      /* Stream socket */
  hints.ai_flags = AI_PASSIVE;          /* Adresse IP joker */
```

```
44    hints.ai_flags |= AI_V4MAPPED|AI_ALL; /* IPv4 remapped en IPv6 */
      hints.ai_protocol = 0;                /* Any protocol */

      s = getaddrinfo(NULL, argv[1], &hints, &result);
      if (s != 0) {
49      fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
        exit(EXIT_FAILURE);
      }

      /* getaddrinfo() retourne une liste de structures d'adresses.
54       On essaie chaque adresse jusqu'a ce que bind(2) reussisse.
         Si socket(2) (ou bind(2)) echoue, on (ferme la socket et on)
         essaie l'adresse suivante. cf man getaddrinfo(3) */
      for (rp = result; rp != NULL; rp = rp->ai_next) {
        /* Creation de la socket */
59      sfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
        if (sfd == -1)
          continue;
        /* Association d'un port a la socket */
        if (bind(sfd, rp->ai_addr, rp->ai_addrlen) == 0)
64        break;              /* Succes */
        close(sfd);
      }
      if (rp == NULL) {       /* Aucune adresse valide */
        perror("bind");
69      exit(EXIT_FAILURE);
      }
      freeaddrinfo(result); /* Plus besoin */

      /* Positionnement de la machine a etats TCP sur listen */
74    listen(sfd, 5);
      printf("Et le serveur attend ...\n");

      /* Boucle Serveur */
      for(;;) {
79      fromlen = sizeof(struct sockaddr_storage);
        ns = accept(sfd, (struct sockaddr *)&from, &fromlen);
        if (ns == -1 ) {
          perror("accept");
          exit(EXIT_FAILURE);
84      }

        /* Reconnaissance de la machine cliente */
        s = getnameinfo((struct sockaddr *)&from, fromlen,
                        host, NI_MAXHOST, NULL, 0, NI_NUMERICHOST);
89      printf("Machine appellante: %s\n", host);

        /* Caracteristiques des sockets, taille des buffers */
        lg = sizeof(rsz);
        if ( getsockopt(sfd, SOL_SOCKET, SO_RCVBUF, &rsz, &lg) == 0 )
94        printf("SO_RCVBUF: %d octets\n", rsz);
        else
          perror("getsockopt SO_RCVBUF");

        lg = sizeof(ssz);
99      if ( getsockopt(sfd, SOL_SOCKET, SO_SNDBUF, &ssz, &lg) == 0 )
          printf("SO_SNDBUF: %d octets\n", ssz);
        else
          perror("getsockopt SO_SNDBUF");

104     /* Boucle de communication */
        for(;;) {
          sleep(3600); /* Ah... dormir une heure! */
          // nread = read(ns, buf, BUFSIZE); ... et le travail en commentaire :-)
        }
109   }
    }
```

Annexe B

Feuille réponse - TP Analyse de protocoles

B.1 Commandes de paramétrage IP des machines terminales

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

B.2 Étude de UDP

1. .....
   .....
2. .....
   .....
3. .....
   .....
4. 
5. .....
   .....
6. .....
   .....
7. .....
   .....

B.3 Étude de TCP

1. .....
.....

2.

3. .....
.....

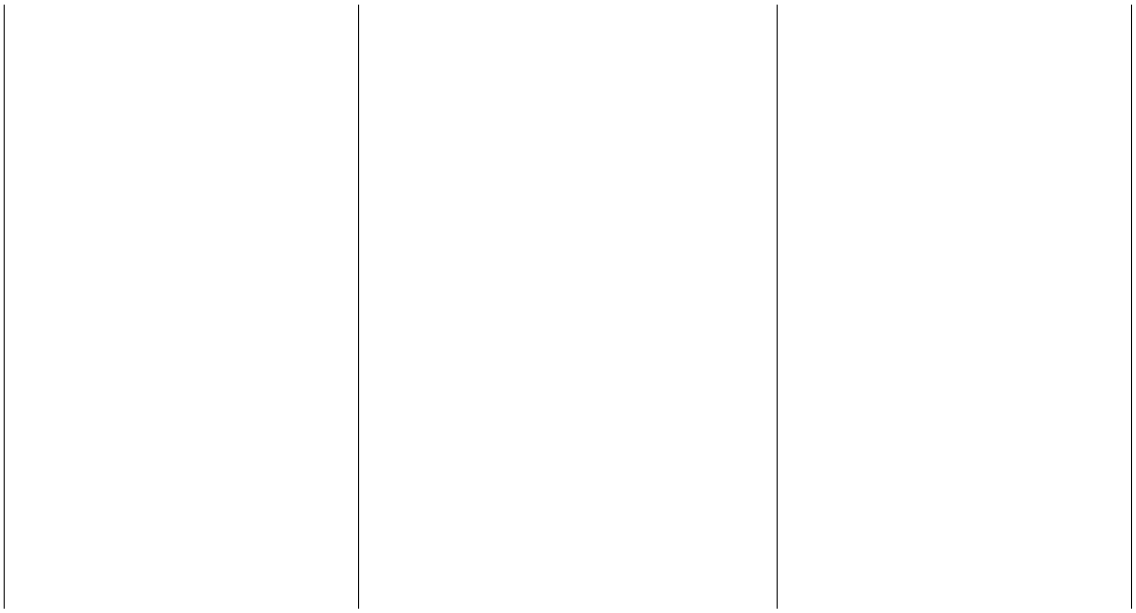
4. .....
.....

5. .....
.....

6. .....
.....

7. .....
.....

8.



B.4 PMTU Discovery

1. 

2. .....
   .....
3. .....
   .....
4. .....
   .....
5. .....
   .....