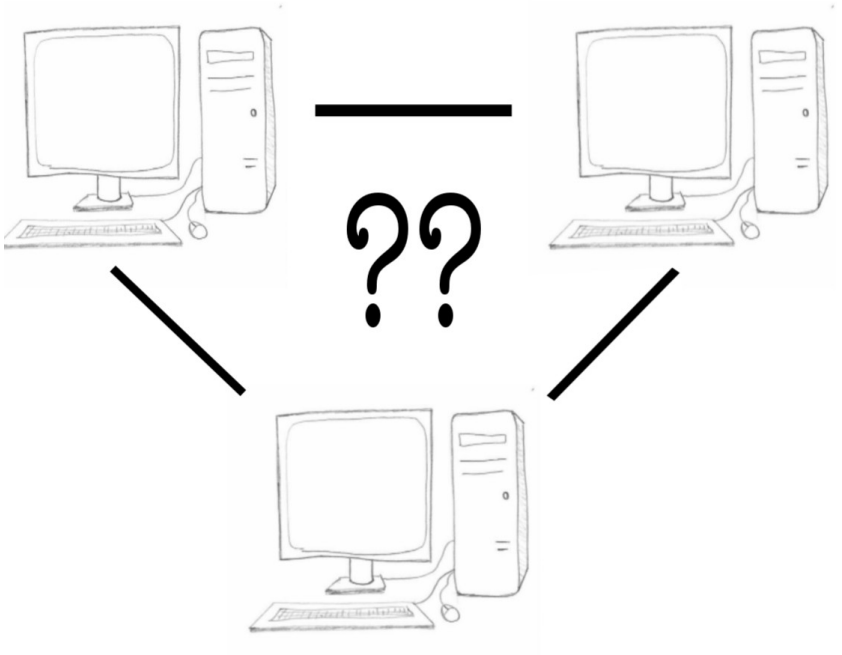
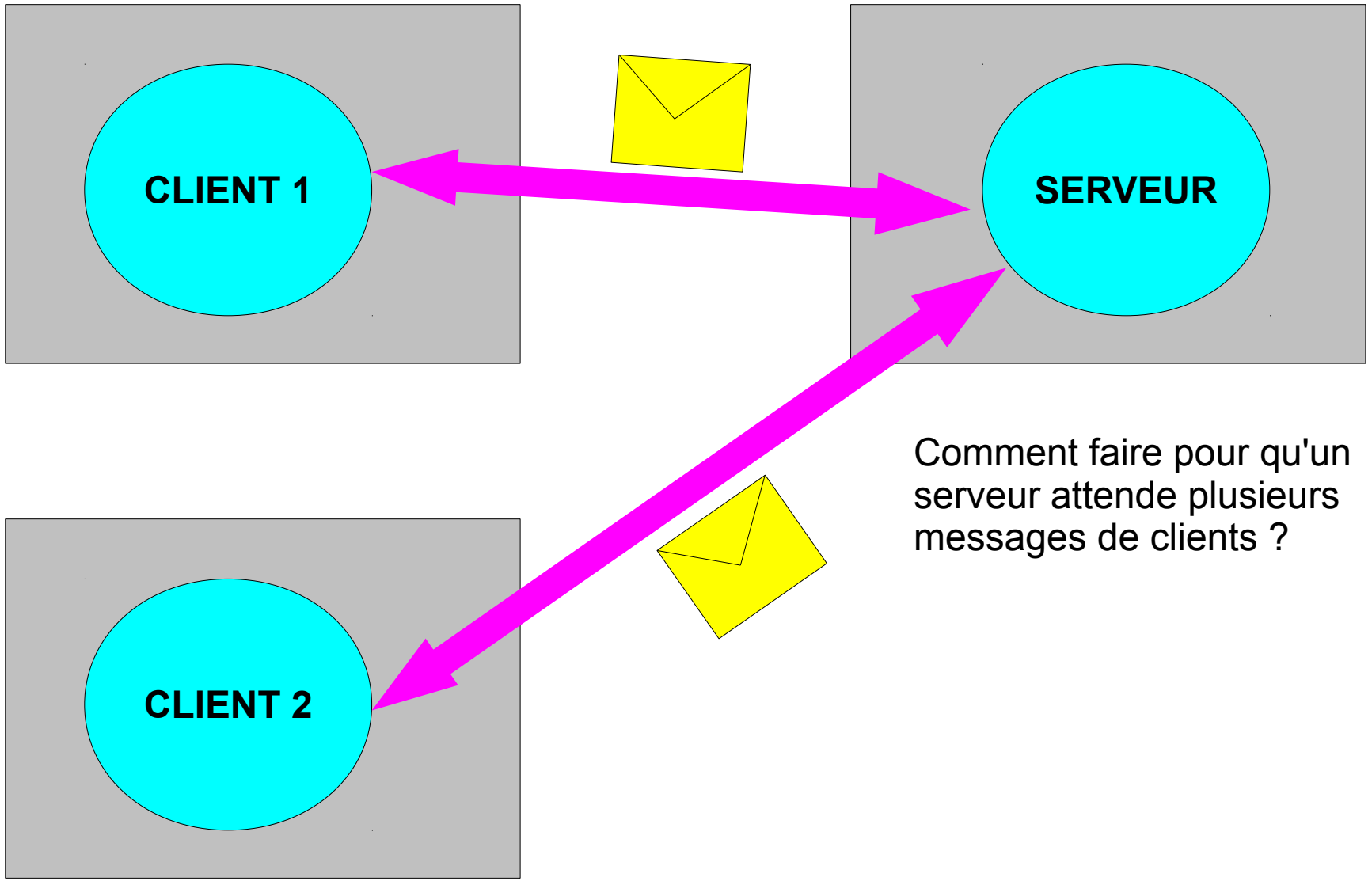
PROGRAMMATION RÉSEAU

Arnaud Sangnier
sangnier@irif.fr

E/S non bloquantes en C



Attente multiple



Comment faire pour qu'un serveur attende plusieurs messages de clients ?

E/S Bloquantes

- Comme en Java, un programme en C peut aussi bloquer
- Par exemple :
  - Un programme qui fait **read** et jamais de données n'arrive
- Exemple de fonctions qui bloquent :
  - **read()**, **recv()**, **recvfrom()**, **accept()**
- En fait, lorsque l'on crée une socket avec socket(), elle est déclarée bloquante par défaut
- Si on veut déclarer une socket comme non bloquante, on peut utiliser la fonction **fcntl** de la façon suivante :
  - **fcntl(sock, F_SETFL, O_NONBLOCK)**
  - **sock** ici est la socket
  - **F_SETFL et O_NONBLOCK** dit de mettre la socket en mode non-bloquant

Problème

- Prenons un programme qui attend des messages UDP sur deux ports différents et qui les affichent
- Comment savoir sur quel port attendre d'abord les messages
- Si on attend d'abord sur le premier port puis ensuite sur le deuxième port et qu'aucun message n'est envoyé sur le premier port alors on a un problème
- La solution inverse qui consiste à attendre sur le deuxième port d'abord a le même inconvénient

Quelles solutions ? (1)

- On peut faire une solution multithreadé
- On crée deux threads
  - Le premier thread attend une donnée sur le port 1
  - Le deuxième thread attend une donnée sur le port 2
- Le parallélisme résout ainsi le problème de blocage que l'on avait avant
- **Avantages** :
  - On sait déjà le faire
- **Inconvénients :**
  - On repousse le problème à gérer correctement la concurrence

Exemple - Récepteur

```
void *recoit_udp(void *arg){
  int sock=*((int *)arg);
  char tampon[100];
  while(1){
    int rec=recv(sock,tampon,100,0);
    tampon[rec]='\0';
    printf("Message recu : %s\n",tampon);
  }
}


int main() {
  int sock1=socket(PF_INET,SOCK_DGRAM,0);
  struct sockaddr_in address_sock1;
  address_sock1.sin_family=AF_INET;
  address_sock1.sin_port=htons(5555);
  address_sock1.sin_addr.s_addr=htonl(INADDR_ANY);
  int sock2=socket(PF_INET,SOCK_DGRAM,0);
  struct sockaddr_in address_sock2;
  address_sock2.sin_family=AF_INET;
  address_sock2.sin_port=htons(5556);
  address_sock2.sin_addr.s_addr=htonl(INADDR_ANY);
  int r=bind(sock1,(struct sockaddr *)&address_sock1,sizeof(struct sockaddr_in));
  if(r==0){
    int r2=bind(sock2,(struct sockaddr *)&address_sock2,sizeof(struct sockaddr_in));
    if(r2==0){
      pthread_t th1,th2;
      pthread_create(&th1,NULL,recoit_udp,&sock1);
      pthread_create(&th2,NULL,recoit_udp,&sock2);
      pthread_join(th1,NULL);
      pthread_join(th2,NULL);
    }
  }
  return 0;
}
```

Exemple - ÉMETTEUR

```
int main(int argc,char *argv[]) {
  if(argc>1){
    int sock=socket(PF_INET,SOCK_DGRAM,0);
    struct addrinfo *first_info;
    struct addrinfo hints;
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;
    hints.ai_socktype=SOCK_DGRAM;
    int r=getaddrinfo("localhost",argv[1],&hints,&first_info);
    if(r==0){
      if(first_info!=NULL){
        struct sockaddr *saddr=first_info->ai_addr;
        char tampon[100];
        int i=0;
        for(i=0;i<=3;i++){
          strcpy(tampon,"MESSAGE ");
          strcat(tampon,argv[1]);
          strcat(tampon," ");
          char entier[3];
          sprintf(entier,"%d",i);
          strcat(tampon,entier);
          sendto(sock,tampon,strlen(tampon),0,saddr,(socklen_t)sizeof(struct
sockaddr_in));
          sleep(1);
        }
      }
    }
  }
  return 0;
}
```

Quelles solutions ? (2)

- Une autre solution consiste à supprimer la concurrence et à rendre les deux sockets UDP sur lesquels on attend les messages non-bloquantes

```
fcntl( sock1, F_SETFL, O_NONBLOCK);
 fcntl( sock2, F_SETFL, O_NONBLOCK);
...
rec1=recv(sock1,tampon,100,0);
printf("Taille de données recues %d\n",rec1);
if(rec1>=0){
  tampon[rec1]='\0';
  printf("Message recu : %s\n",tampon);
}
rec2=recv(sock2,tampon,100,0);
 printf("Taille de données recues %d\n",rec2);
if(rec2>=0){
  tampon[rec2]='\0';
  printf("Message recu : %s\n",tampon);
}
```

- Le problème est que lorsque l'on fait **fcntl(sock, F_SETFL, O_NONBLOCK)** et qu'on fait **recv** sur **sock**. si il n'y a pas de messages, le **recv** renvoie -1

Exemple - Récepteur non satisfaisant

```
int main() {
  int sock1=socket(PF_INET,SOCK_DGRAM,0);
  struct sockaddr_in address_sock1;
  address_sock1.sin_family=AF_INET;
  address_sock1.sin_port=htons(5555);
  address_sock1.sin_addr.s_addr=htonl(INADDR_ANY);
  int sock2=socket(PF_INET,SOCK_DGRAM,0);
  struct sockaddr_in address_sock2;
  address_sock2.sin_family=AF_INET;
  address_sock2.sin_port=htons(5556);
  address_sock2.sin_addr.s_addr=htonl(INADDR_ANY);
  int r=bind(sock1,(struct sockaddr *)&address_sock1,sizeof(struct sockaddr_in));
  if(r==0){
    int r2=bind(sock2,(struct sockaddr *)&address_sock2,sizeof(struct sockaddr_in));
    if(r2==0){
      fcntl( sock1, F_SETFL, O_NONBLOCK);
      fcntl( sock2, F_SETFL, O_NONBLOCK);
      char tampon[100];
      int rec1=0;
      int rec2=0;
      while(1){
        rec1=recv(sock1,tampon,100,0);
        printf("Taille de données recues %d\n",rec1);
        if(rec1>=0){
          tampon[rec1]='\0';
          printf("Message recu : %s\n",tampon);
        }
        rec2=recv(sock2,tampon,100,0);
        printf("Taille de données recues %d\n",rec2);
        if(rec2>=0){
          tampon[rec2]='\0';
          printf("Message recu : %s\n",tampon);
        }
      }
    }
  }
  return 0;
}
```

Comment lire au bon moment ?

- Dans la solution précédente, le problème est que
  - Effectivement on ne bloque plus
  - **MAIS** on teste trop souvent si des données sont disponibles
  - On parle d' **ATTENTE ACTIVE**
- On voudrait pouvoir bloquer si il n'y a aucun message à recevoir et être débloqué dès qu'un message arrive sur l'une des deux sockets
- Comment peut-on faire cela ?
  - On va utiliser la fonction **select**
  - Cette fonction demande au système de réveiller un processus dès qu'une opération sera possible sur un descripteur parmi d'autres
  - Cette fonction permet d'attendre sur tout type de descripteur sockets, tubes, tty etc

La fonction select

- **int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);**
- les **fd_set** sont des ensembles de file descriptor
  - **numfds** est le numéro maximal du file descriptor que l'on souhaite observer plus 1
  - **readfds** est l'ensemble des descripteurs que l'on observe en lecture
  - **writefds** est l'ensemble des descripteurs sur lesquels on attend de pouvoir écrire
  - **exceptfds** est l'ensemble des descripteurs surveiller pour conditions exceptionnelles
  - **timeout** est un temps maximal d'attente, mis à NULL si on ne veut pas de timout
- Le **select** est bloquant et renvoie le nombre de descripteurs sur lequel on peut intervenir

Exemple typique de select

- On veut écouter sur deux descripteurs d1 et d2 en lecture en boucle
-

```
fd_set initial;
int fd_max=0;
FD_ZERO(&initial);
FD_SET(d1,&initial);
fd_max=maximum(fd_max,d1) ; //on suppose qu'il y a une fonction maximum
FD_SET(d2,&initial);
fd_max=maximum(fd_max,d2);
while(1){
  fd_set rdfs; //on initialisera a chaque tour cet ensemle
  FD_COPY(&initial,&rdfs);
  int res=select(fd_max+1, &rdfs, NULL, NULL, NULL) ;
  while(res>0){
     if(FD_ISSET(d1,&rdfs)){ ... ; res-- ;}
     if(FD_ISSET(d2,&rdfs)){ ... ; res-- ;}
  }
}
```

Que faut-il faire pour utiliser select

- Il faut se rappeler du descripteur maximal que l'on manipule
- Il faut créer les ensembles de file descriptor à donner en argument
- Comment manipuler les file descriptor :
  - **FD_SET(int fd, fd_set *set);** Ajoute fd à l'ensemble
  - **FD_CLR(int fd, fd_set *set);** Enlève fd de l'ensemble
  - **FD_ISSET(int fd, fd_set *set);** Renvoie vraie si fd est dans l'ensemble.
  - **FD_ZERO(fd_set *set) ;** Efface tous les éléments de l'ensemble
  - **FD_COPY(fd_set *orig, fd_set *copy);** Copie orig dans copy
- Quand select termine, il renvoie le nombre de descripteurs sur lesquels les opérations ont été réalisées
- Les fd_set ont été modifiées !!!!!! Seuls les descripteurs 'disponibles' sont restées, on peut donc tester avec **FD_ISSET** qui est disponible
- Il faut réinitialiser les fd_set pour un prochain appel à select

À propos du timeout

- **select** n'est pas nécessairement complètement bloquant
- Comme on l'a vu on peut proposer un timeout dans le dernier argument
- Dans ce cas, soit select termine car il y a un événement soit parce que le temps d'attente à dépasser le timeout
- Le timeout est de type **struct timeval *timeout**
    - **struct timeval {**

      int tv_sec; // seconds

      int tv_usec; // microseconds };
- Exemple simple :
    - Programme qui attend que l'on tape un caractère au clavier et si on ne le tape au bout de 2,5 seconds s'arrête

Exemple

```
int main(void)
{
    struct timeval tv;
    fd_set initial;
    int ret=0;
    tv.tv_sec = 2;
    tv.tv_usec = 500000;
    FD_ZERO(&initial);
    FD_SET(STDIN_FILENO, &initial);
    char mess[2];
    while(1){
      fd_set rfds;
      FD_COPY(&initial,&rfds);
      ret=select(STDIN_FILENO+1, &rfds, NULL, NULL, &tv);
      printf("Valeur de retour de select : %d\n",ret);
      if (FD_ISSET(STDIN_FILENO, &rfds)){
        printf("On appuie sur une touche\n");
        read(STDIN_FILENO,mess,1);
        mess[1]='\0';
        printf("Touche %s\n",mess);
      } else {
        printf("Timed out.\n");
        return 0;
      }
    }
    return 0;
}
```

Retour sur notre récepteur

```
int main() {
  int sock1=socket(PF_INET,SOCK_DGRAM,0);
  struct sockaddr_in address_sock1;
  address_sock1.sin_family=AF_INET;
  address_sock1.sin_port=htons(5555);
  address_sock1.sin_addr.s_addr=htonl(INADDR_ANY);
  int sock2=socket(PF_INET,SOCK_DGRAM,0);
  struct sockaddr_in address_sock2;
  address_sock2.sin_family=AF_INET;
  address_sock2.sin_port=htons(5556);
  address_sock2.sin_addr.s_addr=htonl(INADDR_ANY);
  int r=bind(sock1,(struct sockaddr *)&address_sock1,sizeof(struct sockaddr_in));
  if(r==0){
    int r2=bind(sock2,(struct sockaddr *)&address_sock2,sizeof(struct
sockaddr_in));
    if(r2==0){
      fcntl( sock1, F_SETFL, O_NONBLOCK);
      fcntl( sock2, F_SETFL, O_NONBLOCK);
      fd_set initial;
      int fd_max=0;
      FD_ZERO(&initial);
      FD_SET(sock1,&initial);
      if(fd_max<sock1){fd_max=sock1;}
      FD_SET(sock2,&initial);
      if(fd_max<sock2){fd_max=sock2;}
```

Retour sur notre récepteur

```
      char tampon[100];
      int rec1=0;
      int rec2=0;
      while(1){
        fd_set rdfs;
        FD_COPY(&initial,&rdfs);
        int ret=select(fd_max+1, &rdfs, NULL, NULL, NULL);
        while(ret>0){
          if(FD_ISSET(sock1,&rdfs)){
            rec1=recv(sock1,tampon,100,0);
            printf("Taille de données recues %d\n",rec1);
            if(rec1>=0){
              tampon[rec1]='\0';
              printf("Message recu : %s\n",tampon);
            }
            ret--;
          }
          if(FD_ISSET(sock2,&rdfs)){
            rec2=recv(sock2,tampon,100,0);
            printf("Taille de données recues %d\n",rec2);
            if(rec2>=0){
              tampon[rec2]='\0';
              printf("Message recu : %s\n",tampon);
            }
            ret--;
          }
        }
      }
    }
  }
  return 0;
}
```

Autre méthode que select

- Il existe une autre fonction pour mettre en attente des processus sur différents descripteurs
- La fonction **poll**
- La structure est différente du **select**
- Alors que select manipuler un champ de bits (les fd_set) mis à 0 ou 1 selon si on voulait suivre on non le descripteur associé
- On fournit à la fonction poll un tableau de structure à suivre
- **int poll(struct pollfd *ufds, unsigned int nfds, int timeout);**
  - **ufds** est un tableau de struct pollfd
  - **nfds** est est la taille du tableau

Structure manipulée par poll

- **struct pollfd {**

  int fd; //le descripteur

  short events; / l'événement qui nous intéresse

  short revents; //l'événement quand poll retourne};

- Les événements possibles sont :
  - **POLLIN** : lecture ou accept
  - **POLLOUT** : écriture
  - **POLLPRI** : lecture prioritaire
  - **POLLHUP** : déconnexion
  - **POLLERR** : erreur

Retour sur notre récepteur

```
int main() {
  int sock1=socket(PF_INET,SOCK_DGRAM,0);
  struct sockaddr_in address_sock1;
  address_sock1.sin_family=AF_INET;
  address_sock1.sin_port=htons(5555);
  address_sock1.sin_addr.s_addr=htonl(INADDR_ANY);
  int sock2=socket(PF_INET,SOCK_DGRAM,0);
  struct sockaddr_in address_sock2;
  address_sock2.sin_family=AF_INET;
  address_sock2.sin_port=htons(5556);
  address_sock2.sin_addr.s_addr=htonl(INADDR_ANY);
  int r=bind(sock1,(struct sockaddr *)&address_sock1,sizeof(struct sockaddr_in));
  if(r==0){
    int r2=bind(sock2,(struct sockaddr *)&address_sock2,sizeof(struct
sockaddr_in));
    if(r2==0){
      fcntl( sock1, F_SETFL, O_NONBLOCK);
      fcntl( sock2, F_SETFL, O_NONBLOCK);
      struct pollfd p[2];
      p[0].fd=sock1;
      p[0].events=POLLIN;
      p[1].fd=sock2;
      p[1].events=POLLIN;
```

Retour sur notre récepteur

```
      char tampon[100];
      int rec1=0;
      int i;
      while(1){
        int ret=poll(p,2,-1);
        if(ret>0){
          for(i=0;i<2;i++){
            if(p[i].revents==POLLIN){
              rec1=recv(p[i].fd,tampon,100,0);
              printf("Taille de données recues %d\n",rec1);
              if(rec1>=0){
                tampon[rec1]='\0';
                printf("Message recu : %s\n",tampon);
              }
            }

          }
        }
      }
    }
  }
  return 0;
}
```