ICA

Programming with Sockets Part 1

Prof. Jean-Yves Le Boudec
ICA, EPFL

CH-1015 Ecublens
Leboudec@epfl.ch
http://icawww.epfl.ch

Objective

- be able to write C programs with UDP and TCP, using the socket API, for unicast and multicast
- be able to write a "parallel tcp server"

Contents

- A. General
- B. UDP client / server
- C. Parallel Server

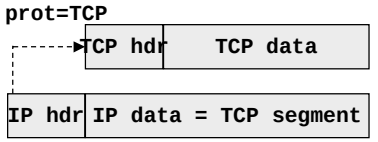
References

- *Socket FAQ* :
  `ftp://rtfm.mit.edu/pub/usenet/news.answers/unix-faq/socket`
- *Processes, fork() under Unix FAQ*
  `ftp://rtfm.mit.edu/pub/usenet/news.answers/programmer/faq`
- TCP/IP, Vol III: Client Server Programming and Applications", 1993, Prentice-Hall

The UDP service

- UDP service interface
  - one message, up to 8K
  - destination address, destination port, source address, source port
- UDP service is message oriented
  - delivers exactly the message or nothing
  - several messages may be delivered in disorder
- UDP used when TCP does not fit
  - short interactions
  - real time, multimedia
  - multicast
- If a UDP message is larger than MTU, then fragmentation occurs at the IP layer

TCP Service: Segments and Bytes



TCP views data as a stream of bytes

- bytes put in packets called TCP segments
  - bytes accumulated in buffer until sending TCP decides to create a segment
  - MSS = maximum "segment" size (maximum data part size)
    - "B sends MSS = 236" means that segments, without header, sent to B should not exceed 236 bytes
  - 536 bytes by default (576 bytes IP packet)
- sequence numbers based on byte counts, not packet counts
- TCP builds segments independent of how application data is broken
  - unlike UDP
- TCP segments never fragmented at source
  - possibly at intermediate points with IPv4
  - where are fragments re-assembled ?

Part A: General: Client Server Model

- processes (for application programs) are associated (dynamically or statically) to port numbers
  - `dest port` used for presenting data to the corresponding program(= demultiplexing at destination)
  - `srce port` stored by destination for responses
- server program
  - program that is ready to receive data at any time
    - on a given port
    - associated with a process running at all times
- client program
  - program that sends data to a server program
    - does not expect to receive data before taking an initiative
- client server computing
  - server programs started in advance
  - client programs (on some other machines) talk to server programs
    - new tasks / processes and/or ports created as result of interaction

Socket Interface

- socket interface is an API:
  - part of UNIX operating system, also in other environments
  - gives access to TCP, UDP, IP and other protocol stacks for the programmer
  - designed to support other protocol types than TCP/IP
    - common interface
- for TCP/IP, three socket types:
  - Stream: TCP
  - Datagram: UDP
  - Raw: IP, ICMP
- a socket is
  - a data structure
  - viewed by UNIX as a file, identified by a socket descriptor (`int`)
  - (IP address, port)

7

Data Structures and Utilities

- addresses

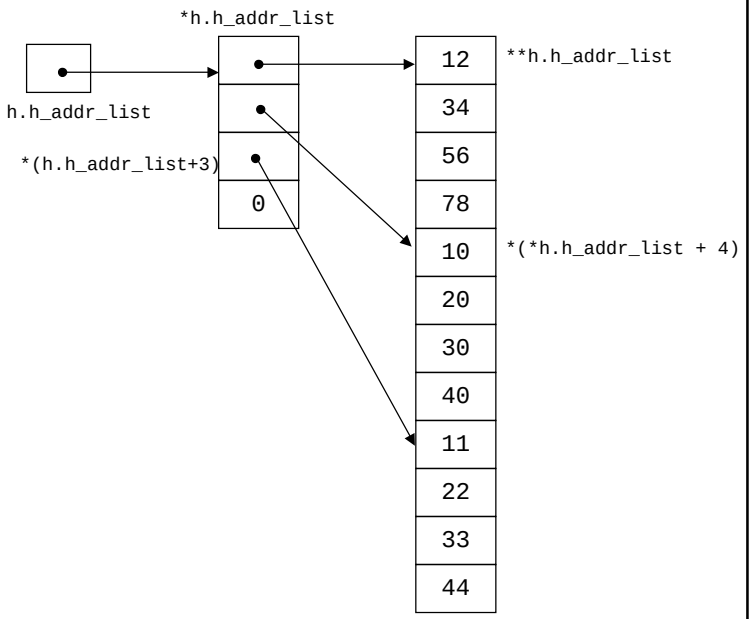
```
struct in_addr {
   u_long   s_addr;
   };

struct sockaddr_in {        /* adresse + port */
   short sin_family;        /* AF_INET */
   u_short sin_port;
   struct in_addr sin_addr;
   char sin_zero[8];        /* unused */
   };

struct in_addr6 {
   u_long   s6_addr[4]
   };

struct sockaddr_in6 {
   short                sin6_family;  /* AF_INET6 */
   u_short              sin6_port;
   u_long               sin6_flowlabel;
   struct in_addr6      sin6_addr;
   };
```

- used by name to address mapping

```
struct hostent {
   char * h_name;         /* host name */
   char **h_aliases;
   int h_addrtype;        /* eg IP */
   int h_length:          /* 4 for IPv4 */
   char **h_addr_list;    /* ends with NULL */
   };
```

8

Illustration



Some Library Procedures

- byte swapping: network order <-> host order

```
u_long  htonl(u_long hostlong);
u_short htons(u_short hostshort);
u_long  ntohl(u_long netlong);
u_short ntohs(u_short netshort);
```

host order: **12 34 56 78 (Motorola)**
78 56 34 12 (Intel)
network order: **12 34 56 78**

- address format translation

```
u_long inet_addr (char* adresseAscii);
char *inet_ntoa(struct in_addr adresse);
```

- map name to address

```
struct hostent *gethostbyname(char* nom);
```


Socket Calls (1: UDP)

- create socket

```
int socket(int family, int type, int protocol)
```

family is `AF_INET AF_NS AF_UNIX AF_INET6`
type is `SOCK_STREAM SOCK_DGRAM SOCK_RAW`
protocol is `0`
return value is socket descriptor or `-1` on error

- bind socket: assign port to a socket

```
int bind(int sd, struct sockaddr* adresse, int longueur);
```

used for specifying a port or obtaining one
and for specifying which interface is used (address field) `AF_INET AF_NS AF_UNIX AF_INET6`

- close socket

```
int close(int sd);
```

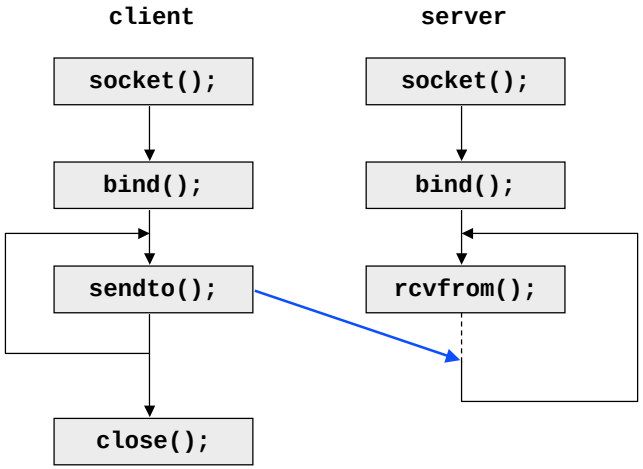
- send or receive for UDP

```
int sendto   (int sd, char* buf, int nbytes, int flags,
              struct sockaddr* adrDest, int longueur);
int recvfrom (int sd, char* buf, int nbytes, int flags,
              struct sockaddr* adrSrce, int* longueur);
```

flags is normally **0**

return value is length of data that was sent or received

Part B: (Very) Simple UDP Client and Server




```
% ./udpClient <destAddr> bonjour les amis
%
```

```
% ./udpServ &
%
```


Client Server Interface

```
/* inet.h */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <stdio.h>

#define SERVER_PORT 1500
#define MAX_MSG 80
#define MAX_FILE 2048
#define TERM_CHAR '$'
```

13

```
/*********************************************/
/*                          udpClient.c      */
/*********************************************/

#include "inet.h"

int main(int argc,  char *argv[]){

  int sd, rc,  i;         // socket descrip.; ret code
  struct sockaddr_in cliAddr, servAddr;
  struct hostent *h;


  // check command line arguments

  if (argc < 3) {
     printf("usage: %s <server> <data1>...<dataN>\n",
            argv[0]);
     exit(1);
     }

  // resolve server name, print result
  // populate address and port

  h = gethostbyname(argv[1]);
  if (h == NULL){
    printf("%s: unknown host '%s'\n", argv[0], argv[1]);
    exit(1);
    }

  printf("%s: trying to send to '%s' (address:   %s )\n",
         argv[0],
         h->h_name,
         inet_ntoa(*(struct in_addr *)h->h_addr_list[0]));

  servAddr.sin_family = h->h_addrtype;
  memcpy((char *) &servAddr.sin_addr.s_addr,h->h_addr_list[0],
         h->h_length);
  servAddr.sin_port = htons (SERVER_PORT);
```

14

```
  // create socket
  sd = socket(AF_INET,SOCK_DGRAM,0);
  if (sd <0) {
     printf("%s: cannot open socket \n",argv[0]);
     exit(1);
     }

 // bind any port number

  cliAddr.sin_family = AF_INET;
  cliAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  cliAddr.sin_port = htons(0);
  //for (i=0; i<8 ; i++) cli_addr.sin_zero[i]='\0';
  rc=bind(sd, (struct sockaddr *) &cliAddr,
              sizeof(cliAddr));
  if (rc<0) {
     printf("%s cannot bind \n", argv[0]);
     exit(1);
     }


  // send data
  for (i=2;i<argc;i++){
    rc = sendto (sd, argv[i], strlen(argv[i])+1,0,
        (struct sockaddr *) &servAddr, sizeof(servAddr));

    if (rc<0){
       printf("%s: cannot send data %d\n",argv[0], i-1);
       close(sd);
       exit(1);
       }


  } // end for

 // close socket and exit
 close(sd);
 exit(0);

}
```

15

```
/*******************************************************/
/*                  udpServ.c                          */
/*******************************************************/

#include "inet.h"

int main(int argc,  char *argv[]){

  int sd, rc, i, n, cliLen;  // socket descriptor
  struct sockaddr_in cliAddr, servAddr;
  char msg[MAX_MSG];


  // create socket
  sd = socket(AF_INET,SOCK_DGRAM,0);
  if (sd <0) {
     printf("%s: cannot open socket \n",argv[0]);
     exit(1);
     }

 // bind server port

  servAddr.sin_family = AF_INET;
  servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servAddr.sin_port = htons(SERVER_PORT);
  rc = bind (sd, (struct sockaddr *) &servAddr,
             sizeof(servAddr));
  if (rc<0) {
     printf("%s cannot bind port number %d \n", argv[0],
             SERVER_PORT);
     exit(1);
     }
```

16

```
// server infinite loop

  while(1){

    // receive
    cliLen = sizeof(cliAddr);
    n = recvfrom(sd, msg, MAX_MSG, 0,
                 (struct sockaddr *) &cliAddr, &cliLen);
    if (n<0){
       printf("%s: cannot receive data \n", argv[0]);
       continue;
       }

    // print message received
    printf("%s: from %s : %s\n",
           argv[0],
           inet_ntoa(cliAddr.sin_addr),
           msg);


  } // end of infinite while


// never reach this line


}
```

17

Socket Calls (2: TCP)

- server
  - tell OS to receive and queue SYN packets

```
int listen(int sd, int queueLength);
```

  - accept connection and create new socket

```
int accept(int sd, struct sockaddr* adrDest, int
    longueur);
```

    returns the new socket descriptor;

- client
  - establish connection to server

```
int connect (int sd, struct sockaddr* adrDest, int
    longueur);
```

- client or server
  - send or receive for TCP (also for UDP, see exercise)

```
int send   (int sd, char* buf, int nBytes, int flags);

int recv (int newSd, char* buf, int nBytes, int flags);
```

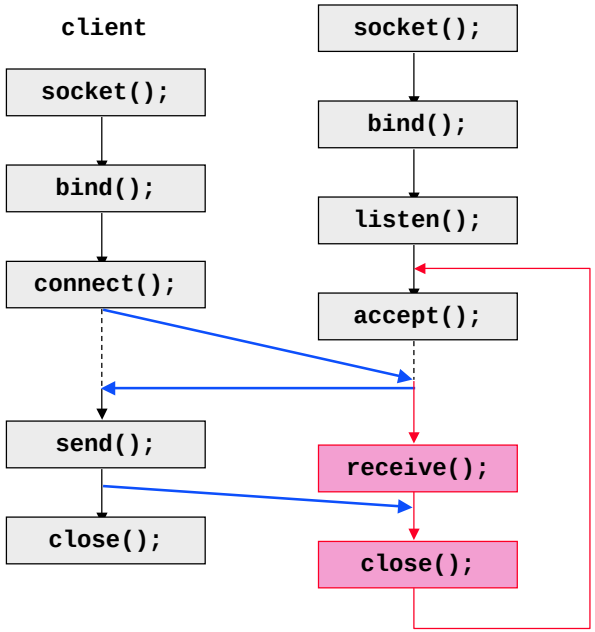
    returns number of bytes received

    0 means connection was closed by other end

    flags is normally 0

18

Part C: TCP client and server




```
% ./tcpClient <destAddr> bonjour les amis
%
```

```
% ./tcpServ &
%
```

```
/*********************************************************/
 /*               tcpClient.c
*/

*********************************************************/

#include "inet.h"

int main(int nbArgPlusUn,  char *mot[]){

  int sd, i;         // socket descriptor
  int rc             // REXXish return code
  struct sockaddr_in cliAddr, servAddr;
  struct hostent *h;


  // check command line arguments

  if (nbArgPlusUn < 3) {
    printf("usage: %s <server> <data1>...<dataN>\n",
          mot[0]);
    exit(1);
    }

  // resolve server name, print result
  // and populate server address and port

  h = gethostbyname(mot[1]);
  if (h == NULL){
    printf("%s: unknown host '%s'\n", mot[0], mot[1]);
    exit(1);
    }

  printf("%s: now preparing to send data to host '%s' \nat
address: %s \n",
         mot[0],
         h->h_name,
         inet_ntoa(*(struct in_addr *) h->h_addr_list[0]));

  servAddr.sin_family = h->h_addrtype;
  memcpy((char *) &servAddr.sin_addr.s_addr, h ->
         h_addr_list[0],
```



```
  // create socket
  sd = socket(AF_INET,SOCK_STREAM,0);
  if (sd <0) {
     printf("%s: cannot open socket \n",mot[0]);
     exit(1);
     }

 // bind any port number
  cliAddr.sin_family = AF_INET;
  cliAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  cliAddr.sin_port = htons(0);
  rc=bind(sd, (struct sockaddr *) &cliAddr,
              sizeof(cliAddr));
  if (rc<0) {
     printf("%s cannot bind \n", mot[0]);
     exit(1);
     }

  // connect to server
  rc = connect (sd, (struct sockaddr *) &servAddr,
                sizeof(servAddr));
  if (rc<0){
     printf("%s: cannot connect \n",mot[0]);
     close(sd);
     exit(1);
     }
  printf("%s: connecting... \n",mot[0]);

  // send arguments one by one
  for (i=2; i < nbArgPlusUn; i++){
     // send data
     rc = send(sd, mot[i], strlen(mot[i])+1, 0);
     if (rc<0){
        printf("%s: cannot send data%d\n",mot[0], i-1);
        close(sd);
        exit(1);
        }
     printf("%s: sent data%d: '%s'\n",mot[0],i, mot[i]);
     }// end for
 // close socket and exit
 close(sd);
 exit(0);
}
```

21

```
/************************************************/
/*                 tcpServer.c                  */
/*                                              */
/*            simple sequential test server     */
/*            connection closed by client       */
/************************************************/

#include "inet.h"

int main(int nbArgPlusUn,  char *mot[]){

  int sd, newSd, rc, i, n, cliLen;
               // socket descriptors and return code
  struct sockaddr_in cliAddr, servAddr;
  char msg[MAX_MSG];


  // create socket
  sd = socket(AF_INET,SOCK_STREAM,0);
  if (sd <0) {
     printf("%s: cannot open socket \n",mot[0]);
     exit(1);
     }

 // bind server port

  servAddr.sin_family = AF_INET;
  servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servAddr.sin_port = htons(SERVER_PORT);
  rc = bind (sd, (struct sockaddr *) &servAddr,
             sizeof(servAddr));
  if (rc<0) {
     printf("%s cannot bind port number %d \n", mot[0],
SERVER_PORT);
     exit(1);
     }

  // tell OS to receive SYN packets on sd
  // sd is an unconnected socket (associated with local
host and port only)
  listen(sd, 5);
```

22

```
// server infinite loop

  while(1){

    // accept one connection from the queue if any
    // create a socket newSd for that connection
    // newSd is connected: associated to source and
destination
    cliLen = sizeof(cliAddr);
    newSd = accept(sd, (struct sockaddr *) &cliAddr,
&cliLen);
    if (newSd<0){
       printf("%s: cannot accept connections \n", mot[0]);
       continue;
       }

    // receive segments

    while (1) {
      n = recv(newSd, msg, MAX_MSG, 0);
      if (n<0) {
         printf("%s: cannot receive data \n", mot[0]);
         }
      else if(n==0) {
         printf("%s: connection closed by client \n",
mot[0]);
         close(newSd);
         break;
         }

       printf("%s: from %s, received %d bytes : %s\n",
           mot[0],
           inet_ntoa(cliAddr.sin_addr),
           n,
           msg);
      } // end of receive segments

  } // end of infinite loop


// never reach this line
}
```

Part D: Multicast

- multicast IP addresses
  - used only with UDP
  - many servers in principle
  - server has to join explicitely
    supported by socket option
  - in **in.h**:

```
struct ip_mreq
{
struct in_addr imr_multiaddr;
                /* IP multicast address of group */
struct in_addr imr_interface;
             /* local IP address of interface */
};
```

  -
```
struct ip_mreq mreq;
rc = setsockopt(sd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
           (void *) &mreq, sizeof(mreq) );
```

  - **IN_MULTICAST(a)** tests whether **a** is a multicast address
  - set `ttl` appropriately

```
% ./mcastClient <destAddr> bonjour les amis
%
```

```
% ./mcastServ <address> &
%
```



```
/*************************************************************/
/*                          mcastClient.c                    */
/*                    multicast test client                  */
/*************************************************************/

#include "inet.h"

int main(int nbArgPlusUn,  char *mot[]){

  int sd, rc,  i;         // socket descriptor and ret code
  unsigned char ttl = 1;  // send multicast with ttl =1 !
  struct sockaddr_in cliAddr, servAddr;
  struct hostent *h;


  // check command line arguments

  if (nbArgPlusUn < 3) {
     printf("usage: %s <server>
<data1>...<dataN>\n",mot[0]);
     exit(1);
     }

  // resolve server name, print result and populate server
  // address and port

  h = gethostbyname(mot[1]);
  if (h == NULL){
    printf("%s: unknown host '%s'\n", mot[0], mot[1]);
    exit(1);
    }

  printf("%s: trying to send data to host '%s' at address:
%s \n",
         mot[0],
         h->h_name,
         inet_ntoa(*(struct in_addr *) h->h_addr_list[0]));

  servAddr.sin_family = h->h_addrtype;
  memcpy((char *) &servAddr.sin_addr.s_addr, h -
>h_addr_list[0],
         h->h_length);
  servAddr.sin_port = htons (SERVER_PORT);
```

25

```
  // check dest addr is multicast;
  if (!IN_MULTICAST(ntohl(servAddr.sin_addr.s_addr))){
     printf("%s: dest addr %s is not multicast \n",mot[0],
            inet_ntoa(servAddr.sin_addr));
     exit(1);
     }



  // create socket
  sd = socket(AF_INET,SOCK_DGRAM,0);
  if (sd <0) {
     printf("%s: cannot open socket \n",mot[0]);
     exit(1);
     }

 // bind any port number

  cliAddr.sin_family = AF_INET;
  cliAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  cliAddr.sin_port = htons(0);

  rc=bind(sd, (struct sockaddr *) &cliAddr,
sizeof(cliAddr));
  if (rc<0) {
    printf("%s cannot bind \n", mot[0]);
    exit(1);
    }

  // set ttl on the socket

  rc = setsockopt(sd, IPPROTO_IP, IP_MULTICAST_TTL, &ttl,
sizeof(ttl));
  if ( rc < 0) {
    printf("%s cannot set ttl = %d IPPROTO_IP,
IP_MULTICAST_TTL \n",
      mot[0], ttl);
    exit(1);
    }
```

26

```
  // send data
  for (i=2;i<nbArgPlusUn;i++){
    rc = sendto (sd, mot[i], strlen(mot[i])+1,0,
        (struct sockaddr *) &servAddr, sizeof(servAddr));

    if (rc<0){
       printf("%s: cannot send data %d\n",mot[0], i-1);
       close(sd);
       exit(1);
       }


  } // end for

 // close socket and exit
 close(sd);
 exit(0);

}
```

27

```
/*******************************************************/
/*                          mcastServ.c                */
/*                                                     */
/*                     multicast test server           */
/*******************************************************/

#include "inet.h"

int main(int nbArgPlusUn,  char *mot[]){

  int sd, rc, i, n, cliLen;
  struct ip_mreq mreq;      // req block for mcast address
  struct sockaddr_in cliAddr, servAddr;
  struct in_addr mcastAddr;
  struct hostent *h;
  char msg[MAX_MSG];


  // check command line arguments

  if (nbArgPlusUn != 2) {
     printf("usage: %s <mcast address>\n",mot[0]);
     exit(1);
     }

  // get multicast address for server to listen to
  h = gethostbyname(mot[1]);
  if (h == NULL){
    printf("%s: unknown group '%s'\n", mot[0], mot[1]);
    exit(1);
    }

  memcpy(&mcastAddr, h ->h_addr_list[0], h->h_length);

  // check dest addr is multicast;
  if (!IN_MULTICAST(ntohl(mcastAddr.s_addr))){
     printf("%s: dest addr %s is not multicast \n",mot[0],
            inet_ntoa(mcastAddr));
     exit(1);
     }
  printf("%s: server ready to listen to %s\n", mot[0],
mot[1]);
```

28

```
  // create socket
  sd = socket(AF_INET,SOCK_DGRAM,0);
  if (sd <0) {
     printf("%s: cannot open socket \n",mot[0]);
     exit(1);
     }

 // bind server port

  servAddr.sin_family = AF_INET;
  servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servAddr.sin_port = htons(SERVER_PORT);
  rc = bind (sd, (struct sockaddr *) &servAddr,
sizeof(servAddr));
  if (rc<0) {
     printf("%s cannot bind port number %d \n", mot[0],
SERVER_PORT);
     exit(1);
     }

  // join multicast group

  mreq.imr_multiaddr.s_addr = mcastAddr.s_addr;
  mreq.imr_interface.s_addr = htonl(INADDR_ANY);
  rc = setsockopt(sd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                  (void *) &mreq, sizeof(mreq) );
  if (rc<0) {
     printf("%s cannot join multicast address %s \n",
mot[0],
             inet_ntoa(mcastAddr));
     exit(1);

     }
  else
   printf("%s now listening to multicast address %s \n",
          mot[0],
          inet_ntoa(mcastAddr));
```

29

```
  // server infinite loop

  while(1){

    // receive
    cliLen = sizeof(cliAddr);
    n = recvfrom(sd, msg, MAX_MSG, 0,
                 (struct sockaddr *) &cliAddr, &cliLen);
    if (n<0){
       printf("%s: cannot receive data \n", mot[0]);
       continue;
       }

    // print message received
    printf("%s: from %s on address %s: %s\n",
           mot[0],
           inet_ntoa(cliAddr.sin_addr),
           mot[1],
           msg);


  } // end of infinite while


// never reach this line


}
```

30

Part E. Parallel Servers

- Sequential Server:
  - handles requests in sequence
  - = iterative
- Parallel Server
  - handles requests in parallel
  - creates a child process or thread
- Exercise
  - write a simple C program that creates a child process, sleeps for 10 seconds, prints a message and its process id. The child process prints a message and its process id.

Solution

```
/*forkEx.c */

#include <stdio.h>
#include <signal.h>

main(int argc, char* argv[]){

int rc;

  printf("%s : I will fork\n",argv[0]);
  rc=fork();
  if (rc >0){
     sleep(10);
     printf("%s [%i]: I am the father\n",
              argv[0],getpid());
     }
  else printf("%s [%i]: I am the son \n",
               argv[0],getpid());

}
```

use the following slide to illustrate the cloning


```
/*forkEx.c */

#include <stdio.h>
#include <signal.h>

main(int argc, char* argv[]){

int rc;

  printf("%s : I will fork\n",argv[0]);
  rc=fork();
  if (rc >0){
     sleep(10);
     printf("%s [%i]: I am the father\n",
              argv[0],getpid());
     }
  else printf("%s [%i]: I am the son \n",
               argv[0],getpid());

}
```


use this duplicate slide to illustrate fork()

How Many Processes are there (1) ?


```
#include <stdio.h>
#include <signal.h>
#include "inet.h"

main(int argc, char* argv[]){

int rc, i;

for(i=0; i<8; i++){

 printf("%s : I will fork\n",argv[0]);
 rc=fork();
 if (rc >0){
    // sleep(10);
    printf("%s [%i]: I am the
  father\n",argv[0],getpid());
    }
  else {
    printf("%s [%i]: I am the son \n", argv[0],getpid());
    }

 } // end for
}
```

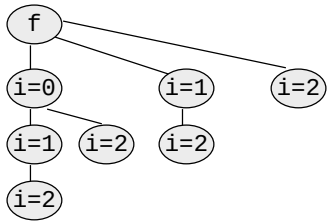
How Many Processes are there (2) ?


```
#include <stdio.h>
#include <signal.h>
#include "inet.h"

main(int argc, char* argv[]){

int rc, i;

for(i=0; i<8; i++){

  printf("%s : I will fork\n",argv[0]);
  rc=fork();
  if (rc >0){
     // sleep(10);
     printf("%s [%i]: I am the
   father\n",argv[0],getpid());
     }
  else {
     printf("%s [%i]: I am the son \n", argv[0],getpid());
     break;
     }

 } // end for
}
```

Solution

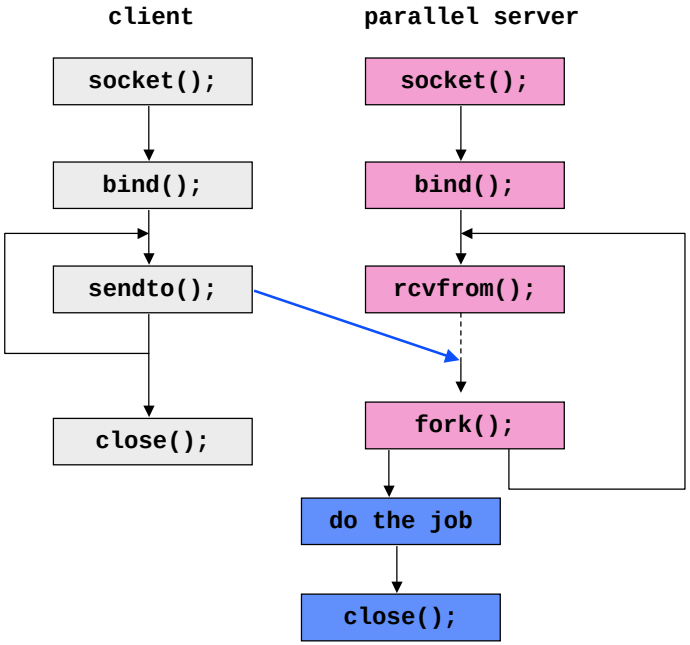
- Example 1: 2^8
  - proof: consider the value of i in the son
    call u(k) the solution for k (in example 1, k=8); on the graph, k=2): we have
    u(k) = u(k-1) + u(k-2) + ... + u(0) + 1
    with u(0) = 1 (look at the subtrees on the fist line below root).

    thus u(k) = 2 u(k-1), cqfd.



- Example 2: 9

Parallel UDP Server




```
% ./udpClient <destAddr> bonjour les amis
%
```

```
% ./udpParServ &
%
```

```
/*********************************************************/
/*             udpParServ.c                              */
/*             simple parallel udp server                */
/*********************************************************/

#include "inet.h"

int main(int nbArgPlusUn,  char *mot[]){

  int sd, rc, i, n, cliLen;
                      // socket descriptor and return code
  struct sockaddr_in cliAddr, servAddr;
  char msg[MAX_MSG];
  int sleepTime;
  int pid;

  // create socket
  sd = socket(AF_INET,SOCK_DGRAM,0);
  if (sd <0) {
     printf("%s: cannot open socket \n",mot[0]);
     exit(1);
     }

 // bind server port

  servAddr.sin_family = AF_INET;
  servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servAddr.sin_port = htons(SERVER_PORT);
  rc = bind (sd, (struct sockaddr *) &servAddr,
            sizeof(servAddr));
  if (rc<0) {
     printf("%s cannot bind port number %d \n",
             mot[0], SERVER_PORT);
     exit(1);
     }

/* avoid zombies */
signal(SIGCHLD, SIG_IGN);
```



```
 // server infinite loop

 while(1){

   // receive
   cliLen = sizeof(cliAddr);
   n = recvfrom(sd, msg, MAX_MSG, 0,
                (struct sockaddr *) &cliAddr, &cliLen);
   if (n<0){
      printf("%s: cannot receive data \n", mot[0]);
      continue;
      }

   // start a new process
   pid = fork();

   if (pid <0){
     printf("%s: cannot receive data \n", mot[0]);
     continue;
     }

   else if (pid==0) {
      // son process
      // do the job
      printf("%s[%d]: processing message '%s' from %s\n",
              mot[0], getpid(), msg,
              inet_ntoa(cliAddr.sin_addr));
      sleepTime = atoi(msg);
      sleep(sleepTime);
      // close socket and die
      close(sd);
      exit(0);
      }

 } // end of infinite while

// never reach this line

}
```

Parallel TCP Server