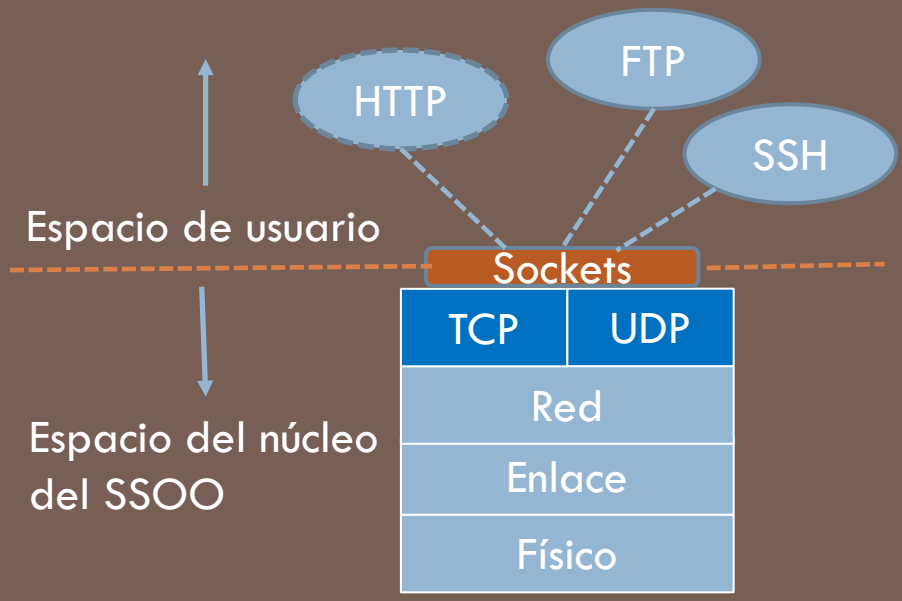
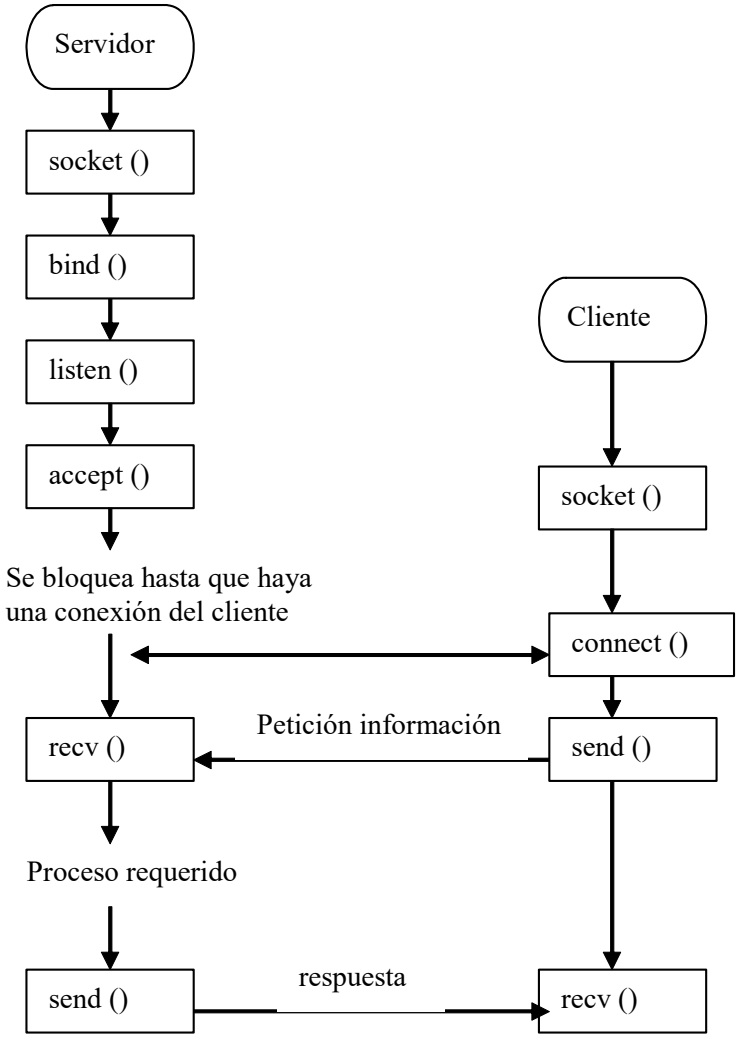


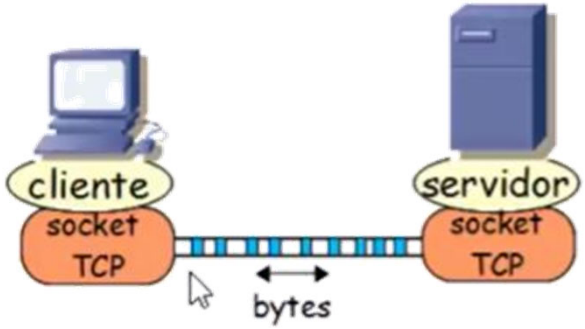
API DE PROGRAMACIÓN EN RED (SOCKETS)

Contenido

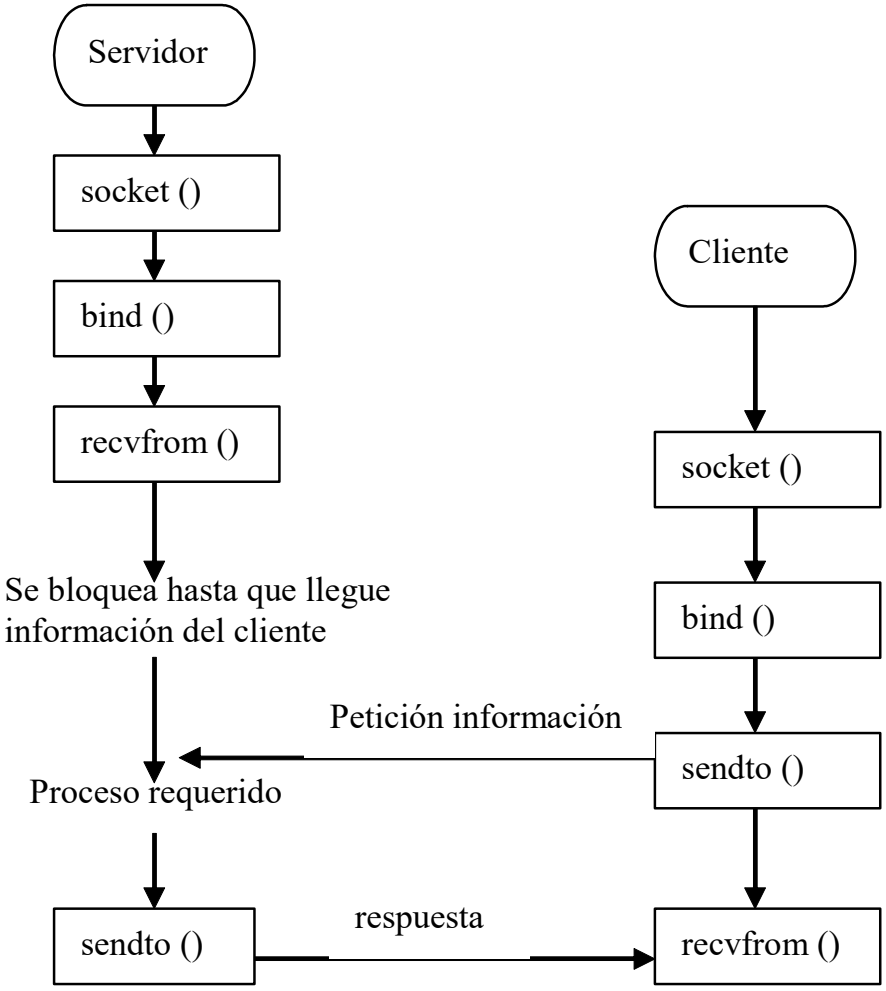
- Los sockets TCP y UDP
- Programas de ejemplo
  - Servidor TCP y UDP
  - Cliente TCP
  - Cliente UDP
  - Consideraciones

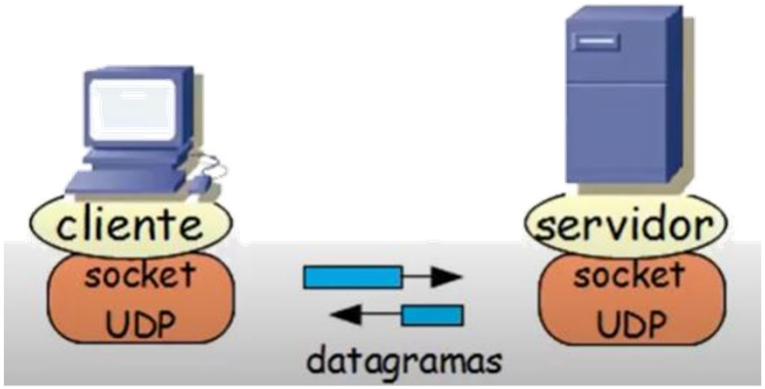
Los sockets de Berkeley: TCP (*stream*)





Los sockets de Berkeley: UDP (*dgram*)





Programas de ejemplo

- servidor.c
  - En TCP devuelve lo que le envía el cliente. Concretamente 1, 2, 3, 4 y 5
  - En UDP devuelve la IP por la que pregunta el cliente
- clientcp.c
  - Envía 1, 2, 3, 4 y 5 y recibe lo mismo
  - `clientcp nombre_servidor`
- clientudp.c
  - Pregunta al servidor por la IP dado un nombre que lee como segundo parámetro
    - clientudp nombre_servidor nombre_cualquier_equipo
    - Ej. `clientudp nogal www.marca.com`

TCP

Ejemplos del servidor y cliente TCP

(servidor.c y clientcp.c)

Servidor TCP (I): *servidor.c*

- Crear el socket de TCP (ls_TCP) – ***Socket*** ()
  - Familia AF_INET (Sockets de Internet con IPv4)
  - Protocolo de transporte de tipo flujo o corriente de datos
    - SOCK_STREAM
  - Protocolo de transporte

```
50    int s_TCP, s_UDP;       /* connected socket descriptor */
93     ls_TCP = socket (AF_INET, SOCK_STREAM, 0);
94     if (ls_TCP == -1) {
95         perror(argv[0]);
96         fprintf(stderr, "%s: unable to create socket TCP\n", argv[0]);
97         exit(1);
98     }
```

Servidor TCP (II): servidor.c

- Inicializar una estructura de tipo *sockaddr_in* con la información del socket
  - Familia de direcciones
  - Dirección IP (la(s) Dir. IP propia)
  - Número de puerto "*bien conocido por los clientes*"
  - *Dos posibles ordenaciones de los octetos* de tipo entero*:*
    - *Primer octeto es el más significativo (BigEndian)*. Por ejemplo Motorola
    - *Primer octeto es el menos significativo (LittleEndian)*. Por ejemplo Intel
  - *Todo lo que se envíe por la red debe seguir la "ordenación de la red" ("BigEndian").*
  - *Métodos de conversión (#include <netinet/in.h>):*
    - *htons(): "Host to Network Short" (short de máquina a short de red).*
    - *htonl(): "Host to Network Long" (long de máquina a long de red).*
    - *ntohs(): "Network to Host Short" (short de la red a short de la máquina).*
    - *ntohl(): "Network to Host Long" (long de red a long de máquina)*

Servidor TCP (II): servidor.c

```
20 #define PUERTO 17278

57     struct sockaddr_in myaddr_in;    /* for local socket address */

69     memset ((char *)&myaddr_in, 0, sizeof(struct sockaddr_in));
```

```
85      myaddr_in.sin_family = AF_INET;
86          /* The server should listen on the wildcard address,
87           * rather than its own internet address.  This is
88           * generally good practice for servers, because on
89           * systems which are connected to more than one
90           * network at once will be able to have one server
91           * listening on all networks at once.  Even when the
92           * host is connected to only one network, this is good
93           * practice, because it makes the server program more
94           * portable.
95           */
96      myaddr_in.sin_addr.s_addr = htonl(INADDR_ANY);
97      myaddr_in.sin_port = htons(PUERTO);
```

Servidor TCP (III): *servidor.c*

- Asociar (**bind**) el socket con la información previamente indicada en la estructura `sockaddr_in`
- Reservar (**listen**) una cola para guardar las peticiones pendientes de aceptación

```
97      if (bind(ls_TCP, (const struct sockaddr *) &myaddr_in, sizeof(struct sockaddr_in)) == -1) {
98          perror(argv[0]);
99          fprintf(stderr, "%s: unable to bind address TCP\n", argv[0]);
100         exit(1);
101     }
102         /* Initiate the listen on the socket so remote users
103          * can connect.  The listen backlog is set to 5, which
104          * is the largest currently supported.
105          */
106     if (listen(ls_TCP, 5) == -1) {
107         perror(argv[0]);
108         fprintf(stderr, "%s: unable to listen on socket\n", argv[0]);
109         exit(1);
110     }
```

Servidor (IV): *servidor.c*

- Convertir el servidor en un proceso demonio (*daemon*)
  - Desvincular el proceso del terminal abierto (`setpgrp`)
  - Crear el proceso que hará las funciones de servidor (`fork`)
  - Ignorar la señal `SIGCHLD` (`sigaction`) para evitar procesos zombi al morir el proceso padre
  - Registrar SIGTERM para la finalización ordenada del programa servidor
  - Bucle infinito para que el proceso esté siempre ejecutándose

```
130     /* Now, all the initialization of the server is
131      * complete, and any user errors will have already
132      * been detected.  Now we can fork the daemon and
133      * return to the user.  We need to do a setpgrp
134      * so that the daemon will no longer be associated
135      * with the user's control terminal.  This is done
136      * before the fork, so that the child will not be
137      * a process group leader.  Otherwise, if the child
138      * were to open a terminal, it would become associated
139      * with that terminal as its control terminal.  It is
140      * always best for the parent to do the setpgrp.
141      */
142     setpgrp();
143
144     switch (fork()) {
145     case -1:        /* Unable to fork, for some reason. */
146         perror(argv[0]);
147         fprintf(stderr, "%s: unable to fork daemon\n", argv[0]);
148         exit(1);
149
150     case 0:     /* The child process (daemon) comes here. */
151
152             /* Close stdin and stderr so that they will not
153              * be kept open.  Stdout is assumed to have been
154              * redirected to some logging file, or /dev/null.
155              * From now on, the daemon will not report any
156              * error messages.  This daemon will loop forever,
157              * waiting for connections and forking a child
158              * server to handle each one.
159              */
160         close(stdin);
161         close(stderr);
162
163             /* Set SIGCLD to SIG_IGN, in order to prevent
164              * the accumulation of zombies as each child
165              * terminates.  This means the daemon does not
166              * have to make wait calls to clean them up.
167              */
168         if ( sigaction(SIGCHLD, &sa, NULL) == -1) {
169             perror(" sigaction(SIGCHLD)");
170             fprintf(stderr,"%s: unable to register the SIGCHLD signal\n", argv[0]);
171             exit(1);
172             }
173
174             /* Registrar SIGTERM para la finalizacion ordenada del programa servidor */
175         vec.sa_handler = (void *) finalizar;
176         vec.sa_flags = 0;
177         if ( sigaction(SIGTERM, &vec, (struct sigaction *) 0) == -1) {
178             perror(" sigaction(SIGTERM)");
179                 public void __cdecl perror (const char *_ErrMsg)M signal\n", argv[0]);
180             exit(1);
181             }
182
183         while (!FIN) {
```

Servidor TCP (V): *servidor.c*

- Aceptar peticiones
  - La función **accept** devuelve un nuevo socket (s_TCP) a través del cual se desarrollará el diálogo con el cliente (multiplexación de conexiones en el mismo número de puerto)
  - La dirección del cliente (IP + puerto efímero) se almacena en una nueva estructura de tipo `sockaddr_in`
  - Para cada cliente que llega se crea un proceso hijo que lo atiende
    - El servidor queda liberado para aceptar nuevos clientes
- El sockets, flujo de datos (*stream*) o asociación queda identificado por: TCP, IP servidor, puerto del servidor, IP cliente, puerto del cliente.

```
216         s_TCP = accept(ls_TCP, (struct sockaddr *) &clientaddr_in, &addrlen);
217         if (s_TCP == -1) exit(1);
218         switch (fork()) {
219             case -1:    /* Can't fork, just exit. */
220                 exit(1);
221             case 0:     /* Child process comes here. */
222                 close(ls_TCP); /* Close the listen socket inherited from the daemon. */
223                 serverTCP(s_TCP, clientaddr_in);
224                 exit(0);
225             default:    /* Daemon process comes here. */
226                     /* The daemon needs to remember
227                      * to close the new accept socket
228                      * after forking the child.  This
229                      * prevents the daemon from running
230                      * out of file descriptor space.  It
231                      * also means that when the server
232                      * closes the socket, that it will
233                      * allow the socket to be destroyed
234                      * since it will be the last close.
235                      */
236                 close(s_TCP);
237             }
```

Servidor TCP (VI): *servidor.c*

- La función serverTCP (I)
  - Dada la dirección IP del cliente obtiene su nombre (`getnameinfo`)
    - En caso de que no sea posible se transforma en el formato decimal punto (`inet_ntop`)
  - Se muestra la dirección IP del cliente (`hostname`), el número de puerto del cliente (ahora convertido al orden del host, `ntohs`) y la hora de llegada

```
286 void serverTCP(int s, struct sockaddr_in clientaddr_in)
287 {
288     int reqcnt = 0;     /* keeps count of number of requests */
289     char buf[TAM_BUFFER];       /* This example uses TAM_BUFFER byte messages. */
290     char hostname[MAXHOST];     /* remote host's name string */
291
292     int len, len1, status;
293     struct hostent *hp;     /* pointer to host info for remote host */
294     long timevar;           /* contains time returned by time() */
295
296     struct linger linger;       /* allow a lingering, graceful close; */
297                                 /* used when setting SO_LINGER */
298
299     /* Look up the host information for the remote host
300      * that we have connected with.  Its internet address
301      * was returned by the accept call, in the main
302      * daemon loop above.
303      */
304
305     status = getnameinfo((struct sockaddr *)&clientaddr_in,sizeof(clientaddr_in),
306                          hostname,MAXHOST,NULL,0,0);
307     if(status){
308             /* The information is unavailable for the remote
309              * host.  Just format its internet address to be
310              * printed out in the logging information.  The
311              * address will be shown in "internet dot format".
312              */
313             /* inet_ntop para interoperatividad con IPv6 */
314             if (inet_ntop(AF_INET, &(clientaddr_in.sin_addr), hostname, MAXHOST) == NULL)
315                 perror(" inet_ntop \n");
316             }
317
318         /* Log a startup message. */
319     time (&timevar);
320         /* The port number must be converted first to host byte
321          * order before printing.  On most hosts, this is not
322          * necessary, but the ntohs() call is included here so
323          * that this program could easily be ported to a host
324          * that does require it.
325          */
326     printf("Startup from %s port %u at %s",
327         hostname, ntohs(clientaddr_in.sin_port), (char *) ctime(&timevar));
```

Servidor TCP (VII): *servidor.c*

- La función serverTCP (II)
  - Configura el socket para un cierre ordenado (setsockopt)

```
333     linger.l_onoff  =1;
334     linger.l_linger =1;
335     if (setsockopt(s, SOL_SOCKET, SO_LINGER, &linger, sizeof(linger)) == -1) {
336         errout(hostname);
337     }
```

Servidor TCP (VIII): *servidor.c*

- La función serverTCP (III)
  - Comienza el diálogo recibiendo datos del cliente (**recv**) a través del socket (s)
    - Los datos son almacenados en una cadena de caracteres (buf)
    - La función devuelve el número de bytes recibidos
  - La sentencia `sleep(1)` representa las tareas que tuviera que hacer el servidor
  - Envía datos (**send**) a través del socket (s)
    - La variable buf representa los datos enviados
    - La función devuelve el número de bytes enviados

```
361     while (len = recv(s, buf, TAM_BUFFER, 0)) {
362         if (len == -1) errout(hostname); /* error from recv */
363             /* The reason this while loop exists is that there
364              * is a remote possibility of the above recv returning
365              * less than TAM_BUFFER bytes.  This is because a recv returns
366              * as soon as there is some data, and will not wait for
367              * all of the requested data to arrive.  Since TAM_BUFFER bytes
368              * is relatively small compared to the allowed TCP
369              * packet sizes, a partial receive is unlikely.  If
370              * this example had used 2048 bytes requests instead,
371              * a partial receive would be far more likely.
372              * This loop will keep receiving until all TAM_BUFFER bytes
373              * have been received, thus guaranteeing that the
374              * next recv at the top of the loop will start at
375              * the begining of the next request.
376              */
377         while (len < TAM_BUFFER) {
378             len1 = recv(s, &buf[len], TAM_BUFFER-len, 0);
379             if (len1 == -1) errout(hostname);
380             len += len1;
381         }
382             /* Increment the request count. */
383         reqcnt++;
384             /* This sleep simulates the processing of the
385              * request that a real server might do.
386              */
387         sleep(1);
388             /* Send a response back to the client. */
389         if (send(s, buf, TAM_BUFFER, 0) != TAM_BUFFER) errout(hostname);
390     }
```

Servidor TCP (y IX): servidor.c

- La función serverTCP (IV)
  - Una vez terminado el diálogo el socket (s) se cierra (`close`)
  - Nuevamente se muestra en pantalla la dirección IP del cliente (`hostname`), el número de puerto del cliente (ahora convertido al orden del host, `ntohs`) y la hora de llegada

```
392         /* The loop has terminated, because there are no
393          * more requests to be serviced.  As mentioned above,
394          * this close will block until all of the sent replies
395          * have been received by the remote host.  The reason
396          * for lingering on the close is so that the server will
397          * have a better idea of when the remote has picked up
398          * all of the data.  This will allow the start and finish
399          * times printed in the log file to reflect more accurately
400          * the length of time this connection was used.
401          */
402     close(s);
403
404         /* Log a finishing message. */
405     time (&timevar);
406         /* The port number must be converted first to host byte
407          * order before printing.  On most hosts, this is not
408          * necessary, but the ntohs() call is included here so
409          * that this program could easily be ported to a host
410          * that does require it.
411          */
412     printf("Completed %s port %u, %d requests, at %s\n",
413         hostname, ntohs(clientaddr_in.sin_port), reqcnt, (char *) ctime(&timevar));
414 }
```

Cliente TCP (I): clientcp.c

- Crear el socket (**socket**) TCP local (s)
  - SOCK_STREAM
- Inicializar la estructura `sockaddr_in` con los datos del servidor al que desea conectarse
  - Familia de direcciones
  - Dirección IP
    - El cliente recoge como argumento el nombre del servidor al que desea conectarse
    - Obtener la IP asociada al nombre (**getaddrinfo**)
  - Número de puerto bien conocido del servidor (`htons`)

```
66          /* Set up the peer address to which we will connect. */
67      servaddr_in.sin_family = AF_INET;
68          /* Get the host information for the hostname that the
69           * user passed in.
70           */
71        memset (&hints, 0, sizeof (hints));
72        hints.ai_family = AF_INET;
73       /* esta función es la recomendada para la compatibilidad con IPv6 gethostbyname queda obsoleta*/
74      errcode = getaddrinfo (argv[1], NULL, &hints, &res);
75      if (errcode != 0)
76      {
77              /* Name was not found.  Return a
78               * special value signifying the
79               * error.
80               */
81          fprintf(stderr, "%s: No es posible resolver la IP de %s\n",
82                  argv[0], argv[1]);
83          exit(1);
84      }
85      else {
86              /* Copy address of host */
87          servaddr_in.sin_addr = ((struct sockaddr_in *) res->ai_addr)->sin_addr;
88      }
89      freeaddrinfo(res);
90  /*  hp = gethostbyname (argv[1]);
91      if (hp == NULL) {
92          fprintf(stderr, "%s: %s not found in DNS system\n",
93                  argv[0], argv[1]);
94          exit(1);
95      }
96      servaddr_in.sin_addr.s_addr = ((struct in_addr *)(hp->h_addr))->s_addr;
97      */
98      servaddr_in.sin_port = htons(PUERTO);
99
100         /* Create the socket. */
101     s = socket (AF_INET, SOCK_STREAM, 0);
102     if (s == -1) {
103         perror(argv[0]);
104         fprintf(stderr, "%s: unable to create socket\n", argv[0]);
105         exit(1);
106     }
```

Cliente TCP (II): clientcp.c

- Conectar con el servidor (**connect**)
- Si se necesita puede obtenerse la información del socket creado (**getsockname**)
  - Rellena una estructura `sockaddr_in` con la información del socket ya conectado (IP + puerto efímero)

clientcp.c

```
107         /* Try to connect to the remote server at the address
108          * which was just built into peeraddr.
109          */
110     if (connect(s, (const struct sockaddr *)&servaddr_in, sizeof(struct sockaddr_in))
111         perror(argv[0]);
112         fprintf(stderr, "%s: unable to connect to remote\n", argv[0]);
113         exit(1);
114     }
115         /* Since the connect call assigns a free address
116          * to the local end of this connection, let's use
117          * getsockname to see what it assigned.  Note that
118          * addrlen needs to be passed in as a pointer,
119          * because getsockname returns the actual length
120          * of the address.
121          */
122     addrlen = sizeof(struct sockaddr_in);
123     if (getsockname(s, (struct sockaddr *)&myaddr_in, &addrlen) == -1) {
124         perror(argv[0]);
125         fprintf(stderr, "%s: unable to read socket address\n", argv[0]);
126         exit(1);
127     }
128
129         /* Print out a startup message for the user. */
130     time(&timevar);
131         /* The port number must be converted first to host byte
132          * order before printing.  On most hosts, this is not
133          * necessary, but the ntohs() call is included here so
134          * that this program could easily be ported to a host
135          * that does require it.
136          */
137     printf("Connected to %s on port %u at %s",
138             argv[1], ntohs(myaddr_in.sin_port), (char *) ctime(&timevar));
139
```

Cliente TCP (y III): clientcp.c

- Diálogo con el servidor enviando datos (**send**) y recibiendo (recv)
- Se puede indicar al servidor la terminación de la fase de envío de datos (**shutdown**)
- Una vez terminado el diálogo se cierra el socket (**close**)

clientcp.c

```
146     for (i=1; i<=5; i++) {
147         *buf = i;
148         if (send(s, buf, 10, 0) != 10) {
149             fprintf(stderr, "%s: Connection aborted on error ",
150                     argv[0]);
151             fprintf(stderr, "on send number %d\n", i);
152             exit(1);
153         }
154     }
155
156         /* Now, shutdown the connection for further sends.
157          * This will cause the server to receive an end-of-file
158          * condition after it has received all the requests that
159          * have just been sent, indicating that we will not be
160          * sending any further requests.
161          */
162     if (shutdown(s, 1) == -1) {
163         perror(argv[0]);
164         fprintf(stderr, "%s: unable to shutdown socket\n", argv[0]);
165         exit(1);
166     }
167
168         /* Now, start receiving all of the replys from the server.
169          * This loop will terminate when the recv returns zero,
170          * which is an end-of-file condition.  This will happen
171          * after the server has sent all of its replies, and closed
172          * its end of the connection.
173          */
174     while (i = recv(s, buf, 10, 0)) {
175         if (i == -1) {
176 errout:         perror(argv[0]);
177             fprintf(stderr, "%s: error reading result\n", argv[0]);
178             exit(1);
179         }
```

UDP

Ejemplos del servidor y cliente UDP

(servidor.c y clientudp.c)

Servidor UDP(I): *servidor.c*

- Crear el socket UDP (**socket**)
  - Familia de direcciones
  - SOCK_DGRAM
- Inicializar una estructura de tipo sockaddr_in
  - Familia de direcciones
  - Dirección IP
  - Número de puerto "bien conocido por los clientes"

```
116     /* Create the socket UDP. */
117     s_UDP = socket (AF_INET, SOCK_DGRAM, 0);
118     if (s_UDP == -1) {
119         perror(argv[0]);
120         printf("%s: unable to create socket UDP\n", argv[0]);
121         exit(1);
122       }
```

```
20 #define PUERTO 17278
```

```
57      struct sockaddr_in myaddr_in;   /* for local socket address */
```

```
69      memset ((char *)&myaddr_in, 0, sizeof(struct sockaddr_in));
```

```
75      myaddr_in.sin_family = AF_INET;
76          /* The server should listen on the wildcard address,
77           * rather than its own internet address.  This is
78           * generally good practice for servers, because on
79           * systems which are connected to more than one
80           * network at once will be able to have one server
81           * listening on all networks at once.  Even when the
82           * host is connected to only one network, this is good
83           * practice, because it makes the server program more
84           * portable.
85           */
86      myaddr_in.sin_addr.s_addr = INADDR_ANY;
87      myaddr_in.sin_port = htons(PUERTO);
```

Servidor UDP(II): *servidor.c*

- Asociar (**bind**) el socket (s_UDP) con la información indicada en la estructura sockaddr_in

```
123         /* Bind the server's address to the socket. */
124     if (bind(s_UDP, (struct sockaddr *) &myaddr_in, sizeof(struct sockaddr_in)) == -1) {
125         perror(argv[0]);
126         printf("%s: unable to bind address UDP\n", argv[0]);
127         exit(1);
128         }
```

Servidor UDP(III): *servidor.c*

- Atender a los clientes
  - La función **recvfrom** además de recibir los datos del cliente (buffer), rellena otra estructura de tipo sockaddr_in (clientaddr_in) con los datos del cliente para poder contestarle (IP + puerto efímero)

```
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
```

```
/* This call will block until a new
* request arrives.  Then, it will
* return the address of the client,
* and a buffer containing its request.
* BUFFERSIZE - 1 bytes are read so that
* room is left at the end of the buffer
* for a null character.
*/
cc = recvfrom(s_UDP, buffer, BUFFERSIZE - 1, 0,
   (struct sockaddr *)&clientaddr_in, &addrlen);
if ( cc == -1) {
    perror(argv[0]);
    printf("%s: recvfrom error\n", argv[0]);
    exit (1);
    }
/* Make sure the message received is
* null terminated.
*/
buffer[cc]='\0';
serverUDP (s_UDP, buffer, clientaddr_in);
```

Servidor UDP(y IV): *servudp.c*

- serverUDP(int s, char * buffer, struct sockaddr_in clientaddr_in)
  - La función **getaddrinfo** dado el nombre obtiene la IP
  - Se envía (**sendto**) la dirección IP solicitada (reqaddr, una estructura de tipo in_addr)

```
454     errcode = getaddrinfo (buffer, NULL, &hints, &res);
455     if (errcode != 0)
456     {
457             /* Name was not found.  Return a
458              * special value signifying the
459              * error.
460              */
461         reqaddr.s_addr = ADDRNOTFOUND;
462     }
463     else {
464             /* Copy address of host into the
465              * return buffer.
466              */
467         reqaddr = ((struct sockaddr_in *) res->ai_addr)->sin_addr;
468
469     }
470      freeaddrinfo(res);
471 /*
472     hp = gethostbyname (buffer);
473     if (hp == NULL) {
474         reqaddr.s_addr = ADDRNOTFOUND;
475     } else {
476         reqaddr.s_addr = ((struct in_addr *)(hp->h_addr))->s_addr;
477     }
478 */
479         /* Send the response back to the
480          * requesting client.  The address
481          * is sent in network byte order. Note that
482          * all errors are ignored.  The client
483          * will retry if it does not receive
484          * the response.
485          */
486     nc = sendto (s, &reqaddr, sizeof(struct in_addr),
487             0, (struct sockaddr *)&clientaddr_in, addrlen);
488     if ( nc == -1) {
489          perror("serverUDP");
490          printf("%s: sendto error\n", "serverUDP");
491          return;
492          }
```

Cliente UDP (I): clientudp.c

- Crear el **socket** UDP local (s)
  - Familia de direcciones
  - SOCK_DGRAM
- Inicializar servaddr_in (estructura de tipo sockaddr_in) con los datos del servidor al que desea conectarse
  - Familia de direcciones
  - Dirección IP
    - El cliente recoge como primer argumento el nombre del servidor al que desea conectarse
    - Obtiene la IP asociada al nombre (**getaddrinfo**)
  - Número de puerto del servidor (**htons**)

```
132     s = socket (AF_INET, SOCK_DGRAM, 0);
133     if (s == -1) {
134         perror(argv[0]);
135         fprintf(stderr, "%s: unable to create socket\n", argv[0]);
136         exit(1);
137     }
```

```
95      servaddr_in.sin_family = AF_INET;
96          /* Get the host information for the server's hostname that the
97           * user passed in.
98           */
99        memset (&hints, 0, sizeof (hints));
100       hints.ai_family = AF_INET;
101
102     /* esta función es la recomendada para la compatibilidad con IPv6 gethostbyname queda obsoleta*/
103     errcode = getaddrinfo (argv[1], NULL, &hints, &res);
104     if (errcode != 0)
105     {
106             /* Name was not found.  Return a
107              * special value signifying the
108              * error.
109              */
110         fprintf(stderr, "%s: No es posible resolver la IP de %s\n",
111                 argv[0], argv[1]);
112         exit(1);
113     }
114     else {
115             /* Copy address of host */
116         servaddr_in.sin_addr = ((struct sockaddr_in *) res->ai_addr)->sin_addr;
117     }
118      freeaddrinfo(res);
119
120     servaddr_in.sin_port = htons(PUERTO);
```

Cliente UDP (II): clientudp.c

- En este caso es obligatorio el uso de la función **bind** para asociar el socket (s) con la información contenida en una estructura de tipo `sockaddr_in`
  - Familia de direcciones (AF_INET)
  - Dirección IP (INADDR_ANY)
  - Número de puerto para el cliente (un 0 significa que el sistema escoja uno libre)
- `bind` rellena la estructura con el número de puerto efímero escogido

clientudp.c

```
119         /* Bind socket to some local address so that the
120          * server can send the reply back.  A port number
121          * of zero will be used so that the system will
122          * assign any available port number.  An address
123          * of INADDR_ANY will be used so we do not have to
124          * look up the internet address of the local host.
125          */
126     myaddr_in.sin_family = AF_INET;
127     myaddr_in.sin_port = 0;
128     myaddr_in.sin_addr.s_addr = INADDR_ANY;
129     if (bind(s, (const struct sockaddr *) &myaddr_in, sizeof(struc
130         perror(argv[0]);
131         fprintf(stderr, "%s: unable to bind socket\n", argv[0]);
132         exit(1);
133     }
```

Cliente UDP (y III): clientudp.c

- Funciones para envío y recepción son
  - `sendto` y `recvfrom`
- El cliente
  - Envía con `sendto`
    - El segundo argumento son los datos que se desean enviar
    - El quinto argumento (servaddr_in) contiene los datos del servidor (IP + puerto "bien conocido")
- Recibe con `recvfrom` recogiendo la dirección del socket remoto
- Hay que habilitar mecanismos de *timeout* y número de reintentos puesto que *recvfrom* es bloqueante y la respuesta puede no llegar nunca

```
while (n_retry > 0) {
    /* Send the request to the nameserver. */
    if (sendto (s, argv[2], strlen(argv[2]), 0, (struct sockaddr *)&servaddr_in,
            sizeof(struct sockaddr_in)) == -1) {
            perror(argv[0]);
            fprintf(stderr, "%s: unable to send request\n", argv[0]);
            exit(1);
        }
    /* Set up a timeout so I don't hang in case the packet
     * gets lost.  After all, UDP does not guarantee
     * delivery.
     */
    alarm(TIMEOUT);
    /* Wait for the reply to come in. */

    if (recvfrom (s, &reqaddr, sizeof(struct in_addr), 0,
                  (struct sockaddr *)&servaddr_in, &addrlen) == -1) {
        if (errno == EINTR) {
                /* Alarm went off and aborted the receive.
                 * Need to retry the request if we have
                 * not already exceeded the retry limit.
                 */
             printf("attempt %d (retries %d).\n", n_retry, RETRIES);
             n_retry--; }
        else  {
            printf("Unable to get response from");
            exit(1); }
    } else {
            alarm(0);
           /* Print out response. */
            if (reqaddr.s_addr == ADDRNOTFOUND)
```

Servidor – Multiplexación de entrada/salida: servidor.c

- Función ***select***
  - Crea el conjunto de sockets o descriptores (macros FD_ZERO y FD_SET) y se selecciona (función `select`) el socket que ha cambiado de estado (habitualmente por la llegada datos)
  - La detección del socket que se ha activado se realiza con la macro FD_ISSET
    - Un bloque para procesar la recepción de datos a través del socket UDP (s_UDP)
      - ServerUDP
    - Otro bloque distinto para procesar la recepción de datos a través del socket TCP (ls_TCP)
      - `accept` crea un nuevo socket (s)
      - serverTCP

```
if ( (numfds = select(getdtablesize(), &readmask, (fd_set *)0, (fd_set *)0, &timeout)) < 0) {
    if (errno == EINTR) {
        FIN=1;
        perror("\nselect failed\n ");
    }
}
else {

 /* Comprobamos si el socket seleccionado es el socket TCP */
 if (FD_ISSET(ls_TCP, &readmask)) {

     s_TCP = accept(ls_TCP, (struct sockaddr *) &clientaddr_in, &addrlen);
     if (s_TCP == -1) exit(1);
     switch (fork()) {
         case -1:    /* Can't fork, just exit. */
             exit(1);
         case 0:     /* Child process comes here. */
             close(ls_TCP); /* Close the listen socket inherited from the daemon. */
             serverTCP(s_TCP, clientaddr_in);
             exit(0);
         default:    /* Daemon process comes here. */

             close(s_TCP);
         }
   } /* De TCP*/
/* Comprobamos si el socket seleccionado es el socket UDP */
if (FD_ISSET(s_UDP, &readmask)) {

     cc = recvfrom(s_UDP, buffer, BUFFERSIZE - 1, 0,
        (struct sockaddr *)&clientaddr_in, &addrlen);
     if ( cc == -1) {
         perror(argv[0]);
         printf("%s: recvfrom error\n", argv[0]);
         exit (1);
         }
     /* Make sure the message received is
     * null terminated.
     */
     buffer[cc]='\0';
     serverUDP (s_UDP, buffer, clientaddr_in);
     }
```

Consideraciones

- Las funciones `send`, `recv`, `sendto` y `recvfrom` permiten enviar y recibir datos de cualquier tipo (`void *`)
  - Se pueden enviar cadenas de caracteres, estructuras, enteros, etc.
  - Sin embargo hay que tener presente que, dado que las arquitecturas de las máquinas cliente y servidor pueden ser distintas, **para garantizar la comunicación y portabilidad del código, se recomienda usar siempre cadenas de caracteres** (o estructuras con miembros de tipo cadena)
  - Estas funciones devuelven el número de caracteres realmente leídos o enviados. Si se desea tratar lo leído como tipo cadena añadir el carácter de fin de cadena.

```
cc = recvfrom(s_UDP, buffer, BUFFERSIZE - 1, 0, (struct sockaddr *)&clientaddr_in, &addrlen);
        if ( cc == -1) {
            perror(argv[0]);
            printf("%s: recvfrom error\n", argv[0]);
            exit (1);
            }
        /* Make sure the message received is null terminated */
        buffer[cc]='\0';
```