Transmisión de Datos y Redes de Computadores II
Práctica 3: Aplicación cliente/servidor: servicio de directorio autenticado

Duración: 2 sesiones

Objetivo

Diseño e implementación de un servicio directorio autentificado y concurrente utilizando la interfaz socket BSD.

Introducción

Recientemente han aparecido una serie de aplicaciones denominadas de "***mensajería instantánea***" (como por ejemplo el Messenger Service MSN http://messenger.msn.com.mx/, ver Fig. 1) cuya funcionalidad principal es la de ofrecer un interfaz sencillo (generalmente en un entorno de ventanas) que facilite el intercambio instantáneo de mensajes sin necesidad explícita de conocer la localización (dirección IP) del usuario destino.

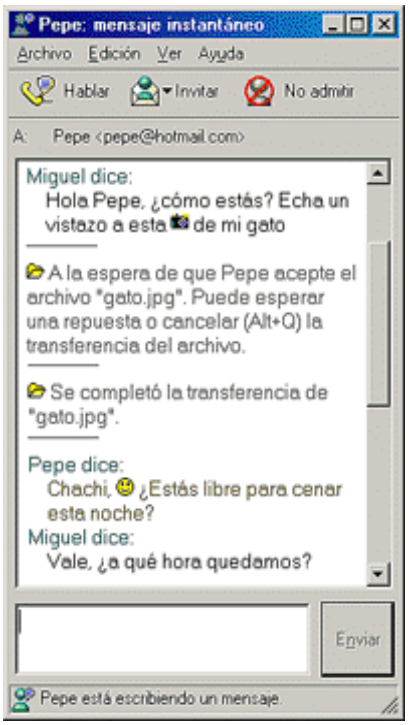

Figura 1: *Messenger Service*

Evidentemente, para conseguir la independencia respecto a la localización del destino, es necesario un ***servicio de directorio*** que proporcione al cliente la dirección IP actual en función de un identificador del usuario final, como por ejemplo su dirección de correo electrónico o su nombre y apellidos. De manera que el cliente, en cualquier momento, pueda saber la localización (dirección IP) y estado (presente o ausente) de un usuario dado.

La práctica que se propone consiste en la realización de una aplicación cliente/servidor que implemente un **servicio de directorio** simplificado mediante el cual, el cliente pueda:

- consultar el estado (presente o ausente) de un usuario
- informar sobre la localización (dirección IP) del usuario actual
- consultar la dirección IP de un usuario presente

La práctica se organiza en tres partes: en la primera el servicio ofrecido será **iterativo**, en una segunda fase, se sugiere la incorporación de un procedimiento de **autenticación**, y por último, en la tercera parte el diseño debe permitir la **concurrencia**.

Realización práctica

Teniendo en cuenta la información proporcionada en los siguientes apartados, y haciendo uso de la bibliografía y documentación recomendada, la realización de la práctica consiste en tres niveles distintos:

1. Realización de un **servicio de directorio iterativo**. Se pide tanto el cliente como el servidor. Se recomienda usar un servicio orientado a conexión. La funcionalidad mínima consiste en la definición de un protocolo que permita:
    a. Comprobar desde el cliente el `estado` (ausente o presente) de un usuario.
    b. Comunicar al servidor el `nombre` y la `localización` (dirección IP) del usuario actual, junto con su `estado`.
    c. Consultar desde el cliente la `localización` (dirección IP) de alguno de los usuarios presentes.
2. La segunda parte de la práctica consiste en la realización de un **servicio de directorio iterativo autenticado**. En este caso, además de la funcionalidad exigida en el nivel 1, se pretende que la comunicación con el servidor en el paso b. sea autenticada. El ususario debe tener un `secreto` establecido previamente en el servidor, tal que antes de realizar la transacción (paso b.) debe enviar la huella digital obtenida a partir de la concatenación de `nombre:reto:secreto`. El servidor debe comprobar la identidad del usuario y en caso afirmativo actualizar el `estado` y la `localización` del usuario.
3. Realización de un **servicio de directotrio concurrente autenticado**. Finalmente, con carácter **no obligatorio**, se sugiere que a la funcionalidad del nivel 2 se añada el carácter de servicio concurrente.

Fundamentos

La programación de aplicaciones sobre TCP/IP se basa en el llamado paradigma o modelo *cliente-servidor*. Básicamente, la idea consiste en que al iniciar un intercambio de información, una de las partes debe "iniciar" el diálogo (cliente) mientras que la otra debe estar indefinidamente preparada a recibir peticiones de establecimiento de dicho diálogo (servidor). Cada vez que un usuario cliente desee entablar un diálogo, primero deberá contactar con el servidor, mandar una petición y posteriormente esperar la respuesta.

Los servidores pueden clasificarse atendiendo a si están diseñados para admitir múltiples conexiones simultáneas (servidores *concurrentes*), por oposición a los servidores que admiten una sola conexión por aplicación ejecutada (llamados *iterativos*). Evidentemente, el diseño de estos últimos será mucho más sencillo que el de los primeros.

Otro concepto importante es la determinación del tipo de interacción entre el cliente y el servidor, que puede ser *orientada a conexión* o *no orientada a conexión*. Éstas corresponden, respectivamente, con los dos protocolos característicos de la capa de transporte: TCP y UDP. TCP (orientado a conexión) garantiza toda la "fiabilidad" requerida para que la transmisión esté libre de errores: verifica que todos los datos se reciben, automáticamente retransmite aquellos que no fueron recibidos, garantiza que no hay errores de transmisión y además, numera los datos para garantizar que se reciben en el mismo orden con el que fueron transmitidos. Igualmente, TCP elimina los datos que por algún motivo aparecen repetidos, realiza un control de flujo para evitar que el emisor envíe más rápido de lo que el receptor puede consumir y, finalmente, informa a las aplicaciones (tanto al cliente como al servidor) si los niveles inferiores de red no pueden entablar la conexión. Por el contrario, UDP (no orientado

a conexión) no introduce ningún procedimiento que garantice la fiabilidad de los datos transmitidos, siendo en este caso responsabilidad de las aplicaciones la realización de los procedimientos necesarios para subsanar cualquier tipo de error.

En el desarrollo de aplicaciones sobre TCP/IP es imprescindible conocer como éstas pueden intercambiar información con los niveles inferiores; es decir, conocer la interfaz con los protocolos TCP o UDP. Esta interfaz es bastante análoga al procedimiento de entrada/salida ordinario en el sistema operativo UNIX que, como se sabe, está basado en la secuencia *abrir-leer/escribir-cerrar*. En particular, la interfaz se basa en la definición de un elemento abstracto denominado "*socket*", concepto muy similar a los descriptores de fichero usados en las operaciones convencionales de entrada/salida en UNIX. Recuérdese que en las operaciones entrada/salida es necesario realizar la apertura del fichero (`open`) antes de que la aplicación pueda acceder a dicho fichero a través del ente abstracto "descriptor de fichero". En la interacción de las aplicaciones con los protocolos TCP o UDP, es necesario que éstas obtengan antes el descriptor o "*socket*", y a partir de ese momento, dichas aplicaciones intercambiarán información con los niveles inferiores (transporte o red) a través del *socket* creado. Una vez creados, los *sockets* pueden ser usados por el servidor para esperar indefinidamente el establecimiento de una conexión (*sockets* pasivos) o, por el contrario, pueden ser usados por el cliente para iniciar la conexión (*sockets* activos).

Estructuras y funciones útiles de la interfaz socket

Para el desarrollo de aplicaciones, la API BSD proporciona una serie de funciones que permiten el manejo de los *sockets*.

Hay que incluir siempre los siguientes ficheros de cabecera:

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/sockets.h>
#include<netinet/in.h>
#include<netdb.h>
```

Repasemos las estructuras de datos importantes. Comenzamos con el problema de los direccionamientos. Los programas de aplicación usan una estructura predefinida cuando necesitan declarar variables que contengan direcciones finales.

La estructura más general para tal fin es la conocida por `sockaddr`, mediante la cual la API gestiona la mayoría de las direcciones

```
struct  sockaddr {
u_short sa_family;   /* familia de direcciones */
char    sa_data[14]; /* hasta 14 bytes de dirección */
}
```

El campo de 2 bytes `sa_family` contiene un identificador de la familia de direcciones, que para el caso de usar TCP/IP debe contener la constante simbólica `AF_INET` (ver el fichero de cabecera `socket.h` para otras familias de direcciones).

No obstante, para mantener los programas portables se recomienda no utilizar la estructura `sockaddr` en las declaraciones. Otra posibilidad que se recomienda es usar la estructura `sockaddr_in` (definida en el fichero de cabecera `#include`

`linux/in.h` incluido desde el fichero `/usr/include/netinet/in.h`) que permite almacenar la identificación de los puntos finales en el siguiente formato:

```
Struct  sockaddr_in {
Short   sin_family;        /* tipo de dirección*/
u_short sin_port;          /* número del puerto*/
struct  in_addr sin_addr; /* dirección IP*/
char    sin_zero[8];       /* no se usa*/
}
```

El campo `sin_family` especifica la familia de protocolos (o arquitectura) a usar; en nuestro caso, como se trata de TCP/IP, dicho campo debe ser igual a la constante `PF_INET`. Igualmente, en el campo `sin_addr` (que es a su vez otra estructura) debe utilizarse sólo el subcampo `u_long s_addr`, que contendrá los 4 bytes de la dirección IP (esto que puede parecer innecesariamente complicado, es así para mantener compatibilidad con versiones antiguas del 4.2BSD). Para aclaración supongamos que definimos la variable `sockname` tipo estructura `sockaddr_in` dentro del `main`:

```
Struct sockaddr_in sockname;
```

Para asignarle un valor deberemos rellenar los siguientes campos:

```
sockname.sin_family=familia_de_protocolos;
sockname.sin_addr.s_addr=dirección_IP;
sockname.sin_port=número_de_puerto;
```

Puede ser útil usar la función `inet_addr`, que realiza una conversión de la cadenas de caracteres de la dirección IP en el formato convencional al formato adecuado para ser utilizado en la estructura `sockaddr_in`.

Otra función de utilidad es `htons`, que cambia el orden de los bytes del entero que se le pasa como argumento. La conversión se realiza del llamado "*host byte order*", en el que el byte LSB precede al MSB, al llamado "*network byte order*" (usado en internet), donde el byte MSB precede al LSB.

La estructura `hostent` definida en el fichero `/usr/include/netdb.h,` contiene entre otras, la direccion IP del *host* en binario:

```
struct hostent {
char   *h_name;       /* nombre del host oficial*/
char   **h_aliases;   /* otros alias*/
int    h_addrtype;    /* tipo de dirección*/
int    h_lenght;      /* longitud de la dirección*/
char   **h_addr_list; /* lista de direcciones*/
}
#define h_addr h_addr_list[0]
```

Asociado con la estructura `hostent` está la función `gethostbyname`, que permite la conversión entre un nombre de host del tipo *hal.ugr.es* a su representación en binario en el campo `h_addr` de la estructura `hostent`.

La estructura `servent` (también definida en el fichero `netdb.h`) contiene, entre otros, como campo el número del puerto con el que se desea comunicar:

```
struct servent {
char   *s_name;     /* nombre oficial del servicio*/
char   **s_aliases; /* otros alias*/
int    s_port;  /* numero del puerto para este servicio*/
char   *s_proto;    /* protocolo a usar*/
}
```

Con la estructura `servent` se relaciona la función `getservbyname` que permite a un cliente o servidor buscar el número oficial de puerto asociado a una aplicación estándar. Ver ejemplos en las páginas 61-63 de [ComerIII].

La función `getprotobyname` permite conocer el número oficial asociado a un protocolo dado, devolviendo el número buscado en la estructura `protoent`:

```
Struct protoent {
char   *p_name;     /* nombre oficial del servicio*/
char   **p_aliases; /* otros alias permitidos*/
int    p_port;      /* número oficial del protocolo*/
}
```

Ejemplo de cliente

El siguiente programa es un ejemplo muy sencillo de un cliente de un servicio de envío de caracteres orientado a conexión, es decir usando TCP. El fichero está disponible en el laboratorio en `/home/redes/practica3/cliente.c`.

```
#include <stdio.h>
#include <malloc.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/time.h>



main()
{
   int sd;
   struct sockaddr_in sockname;
   char buffer[82];

   if((sd=socket(AF_INET,SOCK_STREAM,0))==-1)
        perror("Cliente:Socket"),exit(1);
   sockname.sin_family=AF_INET;
   sockname.sin_addr.s_addr=inet_addr("150.214.60.53");
   sockname.sin_port=htons(15000);
   /*   ^^^^  Sustituir por el puerto correspondiente */
   if(connect(sd,&sockname,sizeof(sockname))==-1)
       perror("Cliente:Connect"),exit(1);
   do {
      puts("Teclee el mensaje a transmitir");
      gets(buffer);
      if(send(sd,buffer,80,0)==-1)
          perror("Cliente:Send"),exit(1);
   } while(strcmp(buffer,"FIN")!=0);
}
```

Ejemplo de servidor iterativo

El siguiente programa es un ejemplo muy sencillo de un servidor ofreciendo un servicio orientado a conexión sobre TCP. El fichero está disponible en el laboratorio en `/home/redes/practica3/servidor.c`.

```
#include <stdio.h>
#include <malloc.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/time.h>

main()
{

      int sd,from_len,new_socket;
      struct sockaddr_in from,sockname;
      char buffer[82];


       if((sd=socket(AF_INET,SOCK_STREAM,0))==-1)
            perror("Servidor:Socket"),exit(1);
       sockname.sin_family=AF_INET;
       sockname.sin_addr.s_addr=INADDR_ANY;
       sockname.sin_port=htons(15000);
      /*     ^^^^   Sustituir por el puerto correspondiente */
       if(bind(sd,&sockname,sizeof(sockname))==-1)
              perror("Servidor:Bind"),exit(1);

       if(listen(sd,1)==-1)
             perror("Servidor1:Listen"),exit(1);

       from_len=sizeof(from);
       if((new_socket=accept(sd,&from,&from_len))==-1)
         perror("Servidor:Accept"),exit(1);

       do{
            if(recv(new_socket,buffer,80,0)==-1)
                  perror("Servidor:Recv"),exit(1);
            printf("El mensaje recibido fue:\n%s\n",buffer);
       }while(strcmp(buffer,"FIN")!=0);
}
```

Terminología y conceptos del procesado concurrente

En este apartado se introducen algunos de los conceptos básicos para el procesado concurrente.

El concepto de proceso

En sistemas de procesado concurrente, la abstracción *proceso* define la unidad fundamental de computación.

Un proceso difiere de un programa porque el concepto de proceso incluye sólo la actividad de ejecución y no el código. En los sistemas de procesado concurrente el sistema puede ejecutar el mismo trozo de código múltiples veces, en el "mismo tiempo". Por supuesto en una arquitectura con un único procesador, la CPU sólo puede ejecutar un proceso en cada instante de tiempo, es el sistema operativo el que hace que aparentemente, desde el punto de vista del usuario, se ejecuten varios procesos simultáneamente.

Función `fork`

```
fork()
```

fork es una función del sistema que se usa en el sistema operativo UNIX para crear un nuevo proceso. La función `fork` divide al programa en ejecución en dos procesos idénticos. Después de ejecutar `fork()` existen dos procesos idénticos, el padre (proceso original) y el hijo (nuevo proceso) ejecutándose en paralelo.

`Fork` devuelve:

- -1 si no se puede crear un proceso hijo.
- 0 al proceso hijo.
- El `PID` del hijo al proceso padre. Esta información es usada para separar los códigos de los dos procesos.

Función `wait`

```
Int wait(int *estado)
```

Esta función devuelve:

- El PID del hijo si ha terminado o ha sido detenido, así como el estado.
- -1 en otro caso (si no hay ningún hijo).
- Si hay varios hijos devuelve el `PID` y el estado del primero que muera.
- Si cuando se invoca a `wait` hay varios hijos muertos, devuelve el `PID` y el estado de uno de ellos.

El siguiente código muestra un ejemplo de uso de las funciones `fork`() y `wait`():

```
if((pid=fork()) == 0) {
   printf("Yo soy el hijo\n");
   exit(0);
}
printf("Soy el padre\n");
```

```
while(wait(&estado) != pid);
printf("Mi hijo ha terminado\n");
```

Procedimiento para limpiar procesos hijos "zombies":

En el programa principal debe ejecutarse

```
(void) signal(SGCHLD, reaper);
```

Siendo

```
int reaper() /* limpia los procesos hijos zombies */
{
   union wait estado;
   while(wait3(&estado,WNOHANG,(struct rusage *)0)>=0);
   (void) signal(SGCHLD, reaper);
}
```

Los servidores concurrentes generan procesos dinámicamente, el sistema operativo UNIX envía una señal al padre cuando existe un proceso hijo. Una vez que se ha salido del proceso hijo, éste se queda en un estado *zombie* hasta que se ejecuta la función del sistema `wait3`. La función del sistema `signal` informa al sistema operativo de que el proceso del servidor maestro puede ejecutar la función `reaper` siempre que reciba una señal que indique que se ha salido de un proceso hijo, esto es cuando `SIGCHLD` tome el valor adecuado. Después de la llamada a `signal`, el sistema automáticamente invoca a `reaper` siempre que el servidor recibe una señal `SIGCHLD`. La función `reaper` llama a `wait3` para finalizar con el proceso hijo. Para asegurar que no se realiza una llamada errónea el programa usa el argumento `WNOHANG`.

Autenticación

La autenticación es uno de los múltiples aspectos que se deben garantizar en un sistema seguro. Este servicio consiste en que cada una de las entidades involucradas esté segura acerca de la identidad de la otra. Es decir, "*cada uno es quien dice ser*".

En una aplicación de servicio de directorio es adecuado ofrecer un servicio de autenticación para que las entidades (y fundamentalmente el cliente) se autentiquen antes de realizar cualquier transacción.

La autenticación puede realizarse de múltiples maneras, si bien en nuestro caso, optaremos por un procedimiento basado en método *hash* y en concreto **MD5**. **MD5** es un algoritmo que toma como entrada un mensaje de longitud arbitraria y genera como salida una *huella digital* (o compendio) de tamaño fijo igual a 128 bits. El algoritmo debe ser tal que sea computacionalmente imposible obtener dos mensajes distintos que generen la misma *huella*.
El código con una implementación de MD5 puede encontrarse en el laboratorio: `/home/redes/practica3/md5.tgz` o en http://www.cs.dartmouth.edu/~jonh/md5/. De hecho, el programa `md5main.c` incluido en las referencias anteriores es un ejemplo de utilización de las funciones correspondientes.

En nuestro sistema, cada usuario tendrá asociada una frase secreta (denominado `secreto`) que se podrá establecer a través de cualquier canal seguro. Cuando un cliente quiera autenticarse, enviará al servidor los siguientes campos:

- `nombre`: identificación (por ejemplo dirección de correo electrónico) del emisor
- `reto`: número aleatorio enviado previamente por el servidor al cliente
- `compendio`: MD5 aplicado a la concatenación de `nombre:reto:secreto`

Toda vez que el compendio sea verificado adecuadamente por el servidor, el usuario `nombre` quedará *autenticado*.

Consideraciones adicionales

- En el laboratorio las direcciones IP se han elegido con libertad absoluta. A cada ordenador se le ha asignado también un nombre, por tanto, cada puesto queda perfectamente definido por su dirección IP o por su nombre.
- Es muy recomendable leer las páginas de manual que considere oportunas. Para ello utilice el comando `man` de LINUX.
- Para compilar sus programas, utilice el compilador de C proporcionado en la instalación actual, utilizando el comando `gcc`.

Bibliografía

[ComerI] D.E. Comer: "*Internetworking with TCP/IP. Vol. I: Principles, Protocols and Architecture".* Prentice-Hall International Editions, second edition, 1991.

[ComerIII] D.E. Comer and D.L. Stevens: "*Internetworking with TCP/IP. Vol III: Client-Server Programming and Applications. BSD Sockets Versión*". Prentice-Hall International Editions, 1993.

[Kurose] Jim Kurose: "*UNIX Network Programming*":

http://www-net.cs.umass.edu/ntu_socket/

[Frost] Jim Frost: "*BSD Sockets: A Quick And Dirty Primer*":

http://world.std.com/~jimf/papers/sockets/sockets.html