INFORME SOBRE ENTREGA SOCKETS SMTP

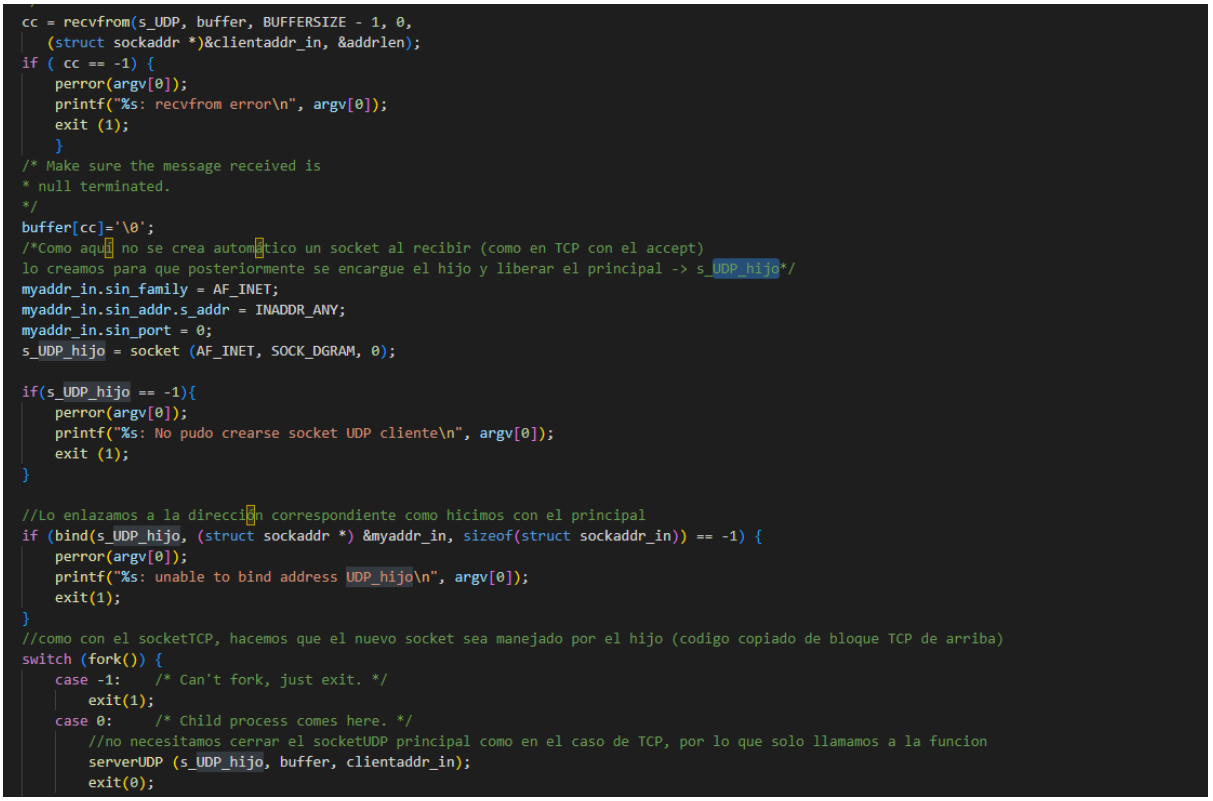
Daniel Lázaro Rubio DNI:70915667C

Germán Francés Tostado DNI:70940996A

ÍNDICE

1. SCRIPT SERVIDOR

Cabe destacar que hemos usado como base los scripts de demostración proporcionados en Diaweb. Se ha usado como número de puerto para el servidor los últimos 4 dígitos del DNI de uno de los componentes del grupo.
La principal modificación que hemos hecho en la función del main es crear un socket UDP_hijo para poder manejar a la vez varias solicitudes de clientes en este protocolo. En TCP se hace mediante el uso de la función accept, creándose así el nuevo socket. Al no estar esa función en UDP hemos creado otro socket de forma manual para manejar las peticiones en la función serverUDP:



```
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';
/*Como aquí no se crea automático un socket al recibir (como en TCP con el accept)
lo creamos para que posteriormente se encargue el hijo y liberar el principal -> s_UDP_hijo*/
myaddr_in.sin_family = AF_INET;
myaddr_in.sin_addr.s_addr = INADDR_ANY;
myaddr_in.sin_port = 0;
s_UDP_hijo = socket (AF_INET, SOCK_DGRAM, 0);

if(s_UDP_hijo == -1){
    perror(argv[0]);
    printf("%s: No pudo crearse socket UDP cliente\n", argv[0]);
    exit (1);
}

//Lo enlazamos a la dirección correspondiente como hicimos con el principal
if (bind(s_UDP_hijo, (struct sockaddr *) &myaddr_in, sizeof(struct sockaddr_in)) == -1) {
    perror(argv[0]);
    printf("%s: unable to bind address UDP_hijo\n", argv[0]);
    exit(1);
}
//como con el socketTCP, hacemos que el nuevo socket sea manejado por el hijo (codigo copiado de bloque TCP de arriba)
switch (fork()) {
    case -1:    /* Can't fork, just exit. */
        exit(1);
    case 0:     /* Child process comes here. */
        //no necesitamos cerrar el socketUDP principal como en el caso de TCP, por lo que solo llamamos a la funcion
        serverUDP (s_UDP_hijo, buffer, clientaddr_in);
        exit(0);
```

En cuanto a las funciones serverTCP y serverUDP, se modificó la funcionalidad que tenían en el script base de Diaweb para adaptarlo al enunciado de la práctica.

A grandes rasgos, se envía al cliente que solicitó el servicio que el servidor está listo para empezar a recibir los parámetros del mensaje SMTP, y en función de las órdenes que recibe envía al cliente un OK en caso de que la orden esté formulada correctamente o un error en caso de que no sea así o sea una orden que no se puede aceptar en ese momento por falta de recepción de otras.

Para asegurar el orden lógico del progama usamos la variable flagFlujo, de manera que por ejemplo no pueda establecer los receptores antes del remitente, o por ejemplo escribir cuerpo de un mensaje que no tenga al menos un receptor.

Uso de variables en las funciones de serverTCP y serverUDP:

```
char ack[BUFFERSIZE]="220 Servicio de transferencia simple de correo preparado\r\n";
char ok[BUFFERSIZE] = "250 OK\r\n";
char dataRes[BUFFERSIZE] = "354 Comenzando con el texto del correo, finalice con .\r\n";
char quitRes[BUFFERSIZE] = "221 Cerrando el servicio\r\n";
char errRes[BUFFERSIZE] = "500 Error de sintaxis\r\n";
int flagFlujo = 1; /* Flag que asegura el correcto flujo de ordenes, para que sean ejecutadas
                      dentro de un orden lógico
                   */
int flagReceptor = 0; /*    Flag para asegurar que haya al menos 1 receptor antes de enviar DATA,
                            no podemos usar flagFlujo ya que puede haber varios Receptores
                      */
int clientPort;
```

Ejemplo de cómo se asegura el orden correcto de las directivas SMTP que se reciben:

```
char *checker = NULL;
// Mientras lea DATA se queda leyendo sin enviar respuesta hasta el .
if(flagFlujo == 4){
    // Está leyendo data, solo va a parar cuando lea un .
    checker = strstr(buf, ".");
    if( checker == buf) {
        // Fin de envío de datos
        if(send (s, ok, BUFFERSIZE, 0) != BUFFERSIZE){
            perror("serverTCP: No se ha podido enviar el mensaje de OK en FINTXT");
            printf("%s: send FINTXT 250 error\n", "serverTCP");
            return;
        }
            fprintf(fPet,"SERVIDOR %s con IP:%s, puerto %u y protocolo TCP- ENVIO: %s\n",hostname,inet_ntoa(clientaddr_in.sin_addr),clientPort,ok);
            flagFlujo = 2;
            flagReceptor = 0;
        continue;

    }
    continue;
}
checker = strstr(buf, "HELO");
if(checker == buf && flagFlujo == 1){
    // Comienza por HELO
    if(send (s, ok, BUFFERSIZE, 0) != BUFFERSIZE){
            perror("serverTCP: No se ha podido enviar el mensaje de OK en HELO");
            printf("%s: send HELO 250 error\n", "serverTCP");
            return;
        }
    fprintf(fPet,"SERVIDOR %s con IP:%s, puerto %u y protocolo TCP- ENVIO: %s\n",hostname,inet_ntoa(clientaddr_in.sin_addr),clientPort,ok);
    flagFlujo=2;
    continue;
}

checker = strstr(buf, "MAIL FROM:");
if(checker == buf && flagFlujo == 2){
    // Comienza por MAIL FROM:
    // Comprobamos que el email sea válido.

    if(validEmail(checker) != 0){ // El email es valido, mandamos ok
        if(send (s, ok, BUFFERSIZE, 0) != BUFFERSIZE){
                perror("serverTCP: No se ha podido enviar el mensaje de OK en MAIL FROM");
                printf("%s: send MAIL 250 error\n", "serverTCP");
                return;
            }
        fprintf(fPet,"SERVIDOR %s con IP:%s, puerto %u y protocolo TCP- ENVIO: %s\n",hostname,inet_ntoa(clientaddr_in.sin_addr),clientPort,ok);
        flagFlujo = 3;
        continue;
    }
```

En el caso de que la orden anteriormente recibida sea DATA, ahora estaremos recibiendo el cuerpo del mensaje, pero dado que nuestro servidor es una versión reducida de SMTP, no tenemos que procesar el texto del mensaje por lo que entraremos en este primer IF. Una vez recibido el mensaje comprueba si es un '.' si lo es, ha terminado el cuerpo del mensaje, por lo que mandará la respuesta 250 OK, sino es un punto, seguirá siendo mensaje por lo que continúa en el bucle de recepción.

Cómo se ve en el bloque de MAIL FROM, hacemos uso de la función validEmail para comprobar si el correo proporcionado por el cliente es válido (que contenga un '@' y no esté vacío a izquierda y derecha). Esto también se comprueba en los RCPT TO:

```c
int validEmail(char str[]){
    /*
        Recibe por argumento la orden leida, y busca un @
        Si existe un @, comprueba que tenga algo por delante y algo por detrás
    */
    int i = 0;
    do{
        i++;
        if(str[i] == '\0') i=0;
        if(str[i] == '@'){
            if(str[i-1] != '\0' && str[i+1] != '\0'){
                return 1;
            }
        }
    }while(i != 0);
    return 0;
}
```

Obviando las distintas funciones a utilizar en función del protocolo (send - sendto o recv - recvfrom) las funciones serverTCP y serverUDP son prácticamente idénticas. El cambio más destacable en serverUDP es el uso de una variable para saber cuando hemos recibido una línea y avisar al cliente de que debe enviar la siguiente, al no tener UDP confirmación. Tras recibir la línea QUIT, se pone el valor de la variable a 0 para romper el bucle:

```c
int flagQuit = 1;

/* Al no tener UDP confirmación, debemos enviar en bucle desde el cliente la misma línea de fichero
hasta que recibamos aquí la línea y confirmemos que la hemos recibido para que pase a enviar
la siguiente en bucle
*/
```

```c
nc = sendto (s, ack, strlen(ack), 0, (struct sockaddr *)&clientaddr_in, addrlen);

if (nc == -1) {
    perror("serverUDP: No se ha podido enviar el mensaje de servicio preparado");
    printf("%s: sendto 220 error\n", "serverUDP");
    return;
}

fprintf(fPet,"SERVIDOR %s con IP:%s, puerto %u y protocolo UDP - ENVIO: %s\n",hostname,inet_ntoa(clientaddr_in.sin_addr),clientPort,ack);

while(flagQuit != 0) {

    nc = recvfrom(s, buffer, BUFFERSIZE - 1, 0,
        (struct sockaddr *)&clientaddr_in, &addrlen);

    if ( nc == -1) {
        perror("serverUDP");
        printf("%s: recvfrom error\n", "serverUDP");
        return;
    }

    buffer[nc]='\0';
```

```c
checker = strstr(buffer, "QUIT");
if(checker == buffer){
    // Comienza por QUIT
    nc = sendto (s, quitRes, strlen(quitRes), 0, (struct sockaddr *)&clientaddr_in, addrlen);

    if (nc == -1) {
        perror("serverUDP: No se ha podido enviar el mensaje de 221 en QUIT");
        printf("%s: sendto QUIT 221 error\n", "serverUDP");
        return;
    }
    fprintf(fPet,"SERVIDOR %s con IP:%s, puerto %u y protocolo UDP - ENVIO: %s\n",hostname,inet_ntoa(clientaddr_in.sin_addr),clientPort,quitRes);
    // activamos flag para que pare de recibir mensajes y se cierre el servicio.
    flagQuit = 0;
    break;
}
```

2. SCRIPT CLIENTE

De nuevo hemos usado como base los scripts proporcionados por Diaweb, aunque lo primero que hubo que hacer para cumplir con los requisitos de la práctica fue juntar todo en un solo script cliente.c. Para conocer qué protocolo se debe usar, hacemos uso de los argumentos de la línea de comandos:

```
if(strcmp(argv[2],"TCP") == 0){

    s = socket (AF_INET, SOCK_STREAM, 0);
```

```
//Socket UDP
else if(strcmp(argv[2],"UDP") == 0){
    /* Create the socket. */
    s = socket (AF_INET, SOCK_DGRAM, 0);
```

Lo primero que se pedía era crear un archivo.txt con el número de puerto efímero asignado para poder imprimir ahí las respuestas del servidor a los envíos del cliente. Para ello hemos usado la función ntohs:

```
printf("Connected to %s on port %u at %s",
        argv[1], ntohs(myaddr_in.sin_port), (char *) ctime(&timevar));


//Abrir el archivo ordenes
FILE *fOrd = fopen(argv[3], "r");
//Archivo log del cliente TCP
char nombreLog[20];
char extension[9] = ".txt";
sprintf(nombreLog,"%u",ntohs(myaddr_in.sin_port));
strcat(nombreLog,extension);

FILE *fLog = fopen(nombreLog,"a"); // para no sobreescribir en caso de que ya exista el archivo
if (fOrd == NULL){
        perror(argv[0]);
        fprintf(stderr, "Error al abrir el archivo %s", argv[3]);
        exit(1);
}
if (fLog == NULL){
        perror(argv[0]);
        fprintf(stderr, "Error al abrir el archivo %s", nombreLog);
        exit(1);
}
```

Para leer las órdenes del archivo que corresponda usamos la función getline, almacenando la línea de texto en la variable line con longitud read. Lo primero que hacemos es insertar los carácteres \r\n:

```
while ((read = getline(&line, &len, fOrd)) != -1) {
    /*
        Enviará las lineas del fichero de ordenes y esperará respuesta.
        Si se envía DATA, no espera respuestas hasta que envíe .
    */

  /*
        Los mensajes en SMTP acaban todos con \r\n, nosotros leemos de ficheros, pero dependiendo
        del SO donde se creó el fichero acabará de una manera u otra.
        Cuando introduzcamos un enter en un archivo acaba así:
        Unix -> \n
        Legacy Mac -> \r
        Windows -> \r\n

        Dado que en read tenemos la longitud de la cadena leida, y sabemos lo anterior:
        En Windows read-2 sería \r y en Unix sería el último caracter de la linea
        asi podemos discriminar entre Windows y Unix/Mac
        si estamos en Unix/Mac, tendremos que escribir en read-1 un \r y en read \n
        para mandar un mensaje terminado en \r\n
    */


        if(line[read-2] != '\r'){
            line[read-1] = '\r';
            line[read] = '\n';
        }


    if (send (s, line, BUFFERSIZE, 0) != BUFFERSIZE) {
        perror(argv[0]);
        fprintf(fLog, "%s: unable to send line on TCP\n", argv[0]);
        exit(1);
    }
```

Si hemos enviado un RCPT TO: comprobamos la respuesta del servidor, si es 250 OK entonces tenemos al menos un receptor válido, por lo que podremos pasar a enviar DATA. Esto lo controlamos mediante el flag dataValido.

```
checker = strstr(line, "RCPT TO:");
if(checker == line){
    // He enviado un RCPT TO, compruebo la respuesta
    checker = strstr(buffer, "250");
    if(checker == buffer){
        dataValido = 1;
    }else{
        fprintf(fLog,"NO VALIDO RCPT TO\n\n");
    }
}
```

Debemos comprobar que la línea que estemos enviando sea DATA, ya que después de esa línea se enviarán otras que no contendrán directivas SMTP (de manera que no van a recibir respuesta del servidor) hasta que se envíe el '.' , por lo que usaremos flagData para seguir la pista de cuándo estamos enviando el cuerpo del mensaje y cuando hemos terminado.

Si hemos enviado la directiva DATA y tenemos al menos un receptor (dataValido = 1), podemos enviar el cuerpo del mensaje (flagData = 1):

```
// Comienza el envío de DATA, espera a recibir la respuesta 354, y luego no espera recepcion hasta que
// envíe un punto
checker = strstr(line, "DATA");
if(checker == line && dataValido == 1){
    flagData = 1;
    dataValido = 0;
}
```

Si hemos enviado el punto, significa que hemos terminado de enviar DATA, por lo que desactivamos el flag y volvemos a poder recibir respuestas del servidor.

Mientras estemos enviando DATA, continuaremos en el bucle leyendo las líneas del fichero de órdenes.

```
// Si envío el . termino de enviar Data y puedo recibir respuesta
checker = strstr(line,".");
if(checker == line ){
    flagData = 0;
}
// Si estoy enviando data, no tengo que esperar a recibir nada hasta que envíe el .
if(flagData != 0){
    continue;
}
```

De nuevo, el tratamiento para TCP y UDP es prácticamente el mismo. Lo que varía es el uso de una cadena con “\r\n” que sirve como petición al servidor para conectarse (en UDP no existe la función connect de TCP) y el uso de un número de intentos dentro de los sendto y recvfrom para que haya un límite de espera en caso de que el servidor no se encuentre operativo o no esté enviando/recibiendo en ese momento:

El mensaje de inicio para la petición de conexión de UDP al servidor será un mensaje vacío con \r\n:

```
char mInicio [4] = "\r\n";
```

```
while (n_retry > 0) {
    /* Send the request to the nameserver. */
    if (sendto (s, mInicio, strlen(mInicio), 0, (struct sockaddr *)&servaddr_in,sizeof(struct sockaddr_in)) == -1) {
            perror(argv[0]);
            fprintf(fLog, "%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, buffer, BUFFERSIZE -1 , 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.
                */
            fprintf(fLog,"attempt %d (retries %d).\n", n_retry, RETRIES);
            n_retry--;
        }
        else  {
            fprintf(fLog,"Unable to get response from");
            exit(1);
        }
    }
```

Si hemos recibido exitosamente la respuesta 220 SMTP preparado del servidor, pasaremos a leer las órdenes del fichero pasado como argumento.

3. PRUEBAS DE FUNCIONAMIENTO

Lanzamiento en nogal del script lanzaServidor.sh:

```
#!/bin/bash
./servidor
./cliente localhost TCP ./ordenes/ordenes.txt &
./cliente localhost TCP ./ordenes/ordenes1.txt &
./cliente localhost TCP ./ordenes/ordenes2.txt &
./cliente localhost UDP ./ordenes/ordenes.txt &
./cliente localhost UDP ./ordenes/ordenes1.txt &
./cliente localhost UDP ./ordenes/ordenes2.txt &
```

```
<ENCINA>/home/i0915667/socketsSMTP$ ./lanza.sh
<ENCINA>/home/i0915667/socketsSMTP$ Connected to localhost on port 55048 at Tue Nov 29 14:02:24 2022
Connected to localhost on port 41764 at Tue Nov 29 14:02:24 2022
Connected to localhost on port 41766 at Tue Nov 29 14:02:24 2022
Connected to localhost on port 43693 at Tue Nov 29 14:02:24 2022
Connected to localhost on port 41772 at Tue Nov 29 14:02:24 2022
Connected to localhost on port 59879 at Tue Nov 29 14:02:24 2022
```

Tras dar un tiempo prudencial a que terminen los clientes TCP deberemos dar un enter para volver a recuperar el uso de la terminal una vez acabados los procesos.

Creación de los .txt correspondientes junto al archivo peticiones.log:

```
<ENCINA>/home/i0915667/socketsSMTP$ ls -l
total 188
-rw-r--r-- 1 i0915667 domainusers   637 nov 29 14:02 41764.txt
-rw-r--r-- 1 i0915667 domainusers   465 nov 29 14:02 41766.txt
-rw-r--r-- 1 i0915667 domainusers   753 nov 29 14:02 41772.txt
-rw-r--r-- 1 i0915667 domainusers   468 nov 29 14:02 43693.txt
-rw-r--r-- 1 i0915667 domainusers   640 nov 29 14:02 55048.txt
-rw-r--r-- 1 i0915667 domainusers   756 nov 29 14:02 59879.txt
-rwxr-xr-x 1 i0915667 domainusers 16956 nov 29 13:07 cliente
-rw-r--r-- 1 i0915667 domainusers 20611 nov 29 13:06 cliente.c
-rw-r--r-- 1 i0915667 domainusers  8908 nov 29 13:07 cliente.o
-rwxr-xr-x 1 i0915667 domainusers   315 nov 29 11:20 lanza.sh
-rw-r--r-- 1 i0915667 domainusers   326 nov 29 11:20 Makefile
drwxr-xr-x 2 i0915667 domainusers  4096 nov 29 11:21 ordenes
-rw-r--r-- 1 i0915667 domainusers 17482 nov 29 14:02 peticiones.log
-rwxr-xr-x 1 i0915667 domainusers 21220 nov 29 13:07 servidor
-rw-r--r-- 1 i0915667 domainusers 31902 nov 29 12:47 servidor.c
-rw-r--r-- 1 i0915667 domainusers 16068 nov 29 13:07 servidor.o
```

Vista de un archivo de petición TCP de cliente:

```
<ENCINA>/home/i0915667/socketsSMTP$ cat 41764.txt
CLIENTETCP antes while - Recibo: 220 Servicio de transferencia simple de correo preparado

CLIENTETCP - Recibo: 250 OK

CLIENTETCP - Recibo: 500 Error de sintaxis

NO VALIDO RCPT TO

CLIENTETCP - Recibo: 250 OK

CLIENTETCP - Recibo: 250 OK

CLIENTETCP - Recibo: 250 OK

CLIENTETCP - Recibo: 354 Comenzando con el texto del correo, finalice con .

CLIENTETCP - Recibo: 250 OK

CLIENTETCP - Recibo: 250 OK

CLIENTETCP - Recibo: 250 OK

CLIENTETCP - Recibo: 354 Comenzando con el texto del correo, finalice con .

CLIENTETCP - Recibo: 250 OK

CLIENTETCP - Recibo: 221 Cerrando el servicio

All done at Tue Nov 29 14:02:40 2022
```

Vista de un archivo de petición UDP de cliente:

```
<ENCINA>/home/i0915667/socketsSMTP$ cat 55048.txt
CLIENTEUDP antes while - Recibo: 220 Servicio de transferencia simple de correo preparado

CLIENTEUDP - Recibo: 250 OK

CLIENTEUDP - Recibo: 500 Error de sintaxis

NO VALIDO RCPT TO

CLIENTEUDP - Recibo: 250 OK

CLIENTEUDP - Recibo: 250 OK

CLIENTEUDP - Recibo: 250 OK

CLIENTEUDP - Recibo: 354 Comenzando con el texto del correo, finalice con .

CLIENTEUDP - Recibo: 250 OK

CLIENTEUDP - Recibo: 250 OK

CLIENTEUDP - Recibo: 250 OK

CLIENTEUDP - Recibo: 354 Comenzando con el texto del correo, finalice con .

CLIENTEUDP - Recibo: 250 OK

CLIENTEUDP - Recibo: 221 Cerrando el servicio

All done at Tue Nov 29 14:02:24 2022
```

Vista del archivo peticiones.log a la última petición registrada:

```
Startup from localhost on IP 127.0.0.1 and port 41772 with protocol TCP at Tue Nov 29 14:02:24 2022
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- ENVIO: 220 Servicio de transferencia simple de correo preparado
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: HELO usal.es
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- ENVIO: 250 OK
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: MAIL FROM <pepe>
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- ENVIO: 500 Error de sintaxis
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: MAIL FROM: <Smith@Alpha.ARPA>
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- ENVIO: 250 OK
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: RCPT TO: <Jones@Beta.ARPA>
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- ENVIO: 250 OK
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: RCPT TO: <Brown@Beta.ARPA>
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- ENVIO: 250 OK
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: DATOS
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- ENVIO: 500 Error de sintaxis
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: DATA
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- ENVIO: 354 Comenzando con el texto del correo, finalice con .
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: Texto de mi correo 2.
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: Linea 2 de mi correo en ordenes2.txt
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: .
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- ENVIO: 250 OK
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: MAIL FROM: <manolo.usal.es>
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- ENVIO: 500 Error de sintaxis
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: MAIL FROM: <manolo@usal.es>
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- ENVIO: 250 OK
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: RCPT TO <cristina@usal.es>
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- ENVIO: 500 Error de sintaxis
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: RCPT TO: <cristina@usal.es>
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- ENVIO: 250 OK
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: DATA
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- ENVIO: 354 Comenzando con el texto del correo, finalice con .
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: Texto de mi correo 22.
```

```
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: DATA
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: Linea 2 de mi correo en ordenes2.txt
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: .
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- ENVIO: 250 OK
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- RECIBO: QUIT
SERVIDOR localhost con IP:127.0.0.1, puerto 41772 y protocolo TCP- ENVIO: 221 Cerrando el servicio
Completed localhost  IP:127.0.0.1 port 41772, on TCP protocol, at Tue Nov 29 14:02:44 2022
```