PRACTICA DE SOCKETS PATATA CALIENTE

La práctica consiste en la implementación de una aplicación red como usuario del nivel de transporte y utilizando la arquitectura o modelo de programación "cliente - servidor"

La estación de trabajo ha sido un sistema operativo Debian GNU/Linux 9(stetch) que es la utilizada en nogal.fis.usal.es. Se han usado sockets de Berjeley y ha sido programado en C.

El juego de la patata caliente consiste en que un grupo de personas se intercambian un item (patata) el cual se va hinchando hasta que explota y el jugador que poseía el item es eliminado. Para poder pasar el item es necesario acertar la pregunta. En esta versión del juego que hemos implementado, solo hay 1 jugador y en vez de una gestión del tiempo, se gestiona el número de intentos al intentar resolver la pregunta.

Los posibles mensajes que se pueden intercambiar son los siguientes:

Cliente	**Servidor**	**Descripción**
	220 Servicio preparado	Respuesta cuando el cliente realiza la conexión
HOLA	250 #	Respuesta a la ordene HOLA con la pregunta y número de intentos
RESPUESTA	354 MAYOR/MENOR#	Respuesta al envío de la orden RESPUESTA indicando si la respuesta correcta es mayor o menor y el número de reintentos que quedan
RESPUESTA	350 ACIERTO	Respuesta al envío de la orden RESPUESTA cuando hay acierto
RESPUESTA	375 FALLO	Respuesta al envío de la orden RESPUESTA cuando el número de intentos es 0
+	250 #	Respuesta a la orden + con la pregunta y número de intentos
ADIOS	221 Cerrando servicio	Respuesta a la orden ADIO
*	500 Error de sintaxis	Respuesta a errores de sintaxis en cualquier orden o secuencias incorrectas

El programa se divide en:

- cliente.c
- servidor.c

CLIENTE.C

Se comunicará con el servidor bien con TCP o con UDP. Este leerá por parámetros el nombre o IP del servidor y el protocolo de transporte TCP o UDP de la siguiente forma:

- cliente "nombre_o_IP_del_servidor" "TCP/UDP" "archivo de órdenes"

Realizará peticiones al servidor como se ha indicado anteriormente y realizará las acciones oportunas para su correcta finalización.

Aspectos relevantes del código

Se comprueban que se le pasan 4 parámetros "cliente, servidor, tipo de conexión y archivo de órdenes" y que estos estén en ese orden, en el caso de que sea erróneo, se imprime como debería ser la llamada y devolvemos 1.

```
int main(int argc, char const *argv[]) {
    if (argc != 4) {
        printf("Uso: ./cliente <servidor> <TCP o UDP> <ordenes.txt o
ordenes1.txt o ordenes2.txt>\n");
        return 1;
    } else if(strcmp(argv[3], "ordenes.txt") != 0 && strcmp(argv[3],
"ordenes1.txt") != 0 && strcmp(argv[3], "ordenes2.txt") != 0) {
        printf("Uso: ./cliente <TCP o UDP> <ordenes.txt o ordenes1.txt o
ordenes2.txt>\n");
        return 1;
    } else if (strcmp(argv[2], "TCP") == 0) {
        const char *nombreArchivo = argv[3];
        const char *hostname = argv[1];
        return clienteTCP(nombreArchivo, hostname);
    } else if (strcmp(argv[2], "UDP") == 0) {
        const char *nombreArchivo = argv[3];
        const char *hostname = argv[1];
        return clienteUDP(nombreArchivo, hostname);
    } else {
        printf("Uso: ./cliente <TCP o UDP> <ordenes.txt o ordenes1.txt o
ordenes2.txt>\n");
        return 1;
    }
}
```

Al cliente TCP le pasamos el fichero de las órdenes y el servidor al que se quiere conectar. Lo primero que hacemos es declarar y darles valores a las estructuras de dirección y al socket que se utilizará, comprobando en todo momento si ocurre algún fallo. Después abrimos el archivo de órdenes, y entramos en un bucle que finalizará cuando el archivo acabe de leerse. Al leer la línea del archivo, nos aseguramos de que este finalice siempre en "\r\n" ya que, sin esto, el servidor nunca reconocerá la orden correctamente y enviamos el mensaje con "send". Vaciamos el array de caracteres de la respuesta para

evitar errores y recibimos la respuesta del servidor con "recv", si esta respuesta es "221 Cerrando el servicio\r\n" finalizaremos el servicio devolviendo 0.

```
int clienteTCP(const char *fichero, const char *hostname) {
    struct sockaddr_in dirLocal;
    struct addrinfo hints, * resultados;
    char solicitud[MAX], respuesta[MAX];
    signal(SIGINT, handler);
    s = socket(AF_INET, SOCK_STREAM, 0);
    if(s == -1) {
        perror("Error al crear el socket");
        return 1;
    }
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;
    if(getaddrinfo(hostname, NULL, &hints, &resultados) != 0) {
        perror("Error al obtener la direccion");
        close(s);
        return 1;
    }
    bzero(&dirLocal, sizeof(dirLocal));
    dirLocal.sin_family = AF_INET;
    dirLocal.sin_addr = ((struct sockaddr_in *) resultados->ai_addr)->sin_addr;
    dirLocal.sin_port = htons(PORT);
    if(connect(s, (struct sockaddr *)&dirLocal, sizeof(dirLocal)) == -1) {
        perror("Error al conectar");
        close(s);
        return 1;
    }
    FILE *f = fopen(fichero, "r");
    if (f == NULL) {
        printf("error al abrir el archivo");
        return 1;
    }
    while(fgets(solicitud, sizeof(solicitud), f) != NULL) {
        int longitud = strlen(solicitud);
        if (longitud > 0 && solicitud[longitud - 1] != '\n') {
            solicitud[longitud] = '\r';
            solicitud[longitud + 1] = '\n';
            solicitud[longitud + 2] = '\0'; // Asegurarse de que la cadena esté
terminada
        }
        if(send(s, solicitud, sizeof(char) * MAX, 0) == -1) {
            perror("Error al enviar la solicitud");
            close(s);
            return 1;
        }
        for (int i = 0; i < MAX; i++) {
            respuesta[i] = '\0';
        }
        if(recv(s, respuesta, sizeof(char) * MAX, 0) == -1) {
```

```
            perror("Error al recibir la respuesta");
            close(s);
            return 1;
        }

        if(strcmp(respuesta, "221 Cerrando el servicio\r\n") == 0) {
            return 0;
        }
    }
}
```

Al cliente UPD, (al igual que al TCP) le pasamos el fichero de las órdenes y el servidor al que se quiere conectar. Lo primero que hacemos es declarar y darles valores a las estructuras de dirección y al socket que se utilizará, comprobando en todo momento si ocurre algún fallo. Después abrimos el archivo de órdenes, y entramos en un bucle que finalizará cuando el archivo acabe de leerse. Al leer la línea del archivo, nos aseguramos de que este finalice siempre en "\r\n" ya que, sin esto, el servidor nunca reconocerá la orden correctamente y enviamos el mensaje con "sendto". Recibimos la respuesta del servidor con "recvform", si esta respuesta es "221 Cerrando el servicio\r\n" finalizaremos el servicio devolviendo 0.

```
int clienteUDP(const char *fichero, const char *hostname) {
    struct sockaddr_in dirLocal;
    struct sockaddr_in dirRemota;
    struct addrinfo hints, * resultados;
    char solicitud[MAX], respuesta[MAX];
    signal(SIGINT, handler);
    s = socket(AF_INET, SOCK_DGRAM, 0);
    if(s == -1) {
        perror("Error al crear el socket");
        return 1;
    }
    bzero(&dirLocal, sizeof(dirLocal));
    bzero(&dirRemota, sizeof(dirRemota));
    dirLocal.sin_family = AF_INET;
    dirLocal.sin_addr.s_addr = htonl(INADDR_ANY);
    dirLocal.sin_port = htons(0);
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;
    if(getaddrinfo(hostname, NULL, &hints, &resultados) != 0) {
        perror("Error al obtener la direccion");
        close(s);
        return 1;
    }
    dirRemota.sin_family = AF_INET;
    dirRemota.sin_addr = ((struct sockaddr_in *) resultados->ai_addr)-
>sin_addr;
```

```
    dirRemota.sin_port = htons(PORT);
    freeaddrinfo(resultados);
    if(bind(s, (struct sockaddr *)&dirLocal, sizeof(struct sockaddr_in)) == -1)
{
        perror("Error al hacer bind");
        close(s);
        return 1;
    }
    socklen_t tamDir = sizeof(struct sockaddr_in);
    FILE *f = fopen(fichero, "r");
    if (f == NULL) {
        printf("error al abrir el archivo");
        return 1;
    }
    while(fgets(solicitud, sizeof(solicitud), f) != NULL) {
        int longitud = strlen(solicitud);
        if (longitud > 0 && solicitud[longitud - 1] != '\n') {
            solicitud[longitud] = '\r';
            solicitud[longitud + 1] = '\n';
            solicitud[longitud + 2] = '\0'; // Asegurarse de que la cadena esté
terminada
        }
        if(sendto(s, solicitud, sizeof(char) * MAX, 0, (struct sockaddr
*)&dirRemota, sizeof(dirRemota)) == -1) {
            perror("Error al enviar la solicitud");
            close(s);
            return 1;
        }
         if(recvfrom(s, respuesta, (sizeof(char) * MAX), 0, (struct sockaddr
*)&dirRemota, &tamDir) == -1) {
            perror("Error al recibir la respuesta");
            close(s);
            return 1;
        }

        if(strcmp(respuesta, "221 Cerrando el servicio\r\n") == 0) {
            return 0;
        }
    }
}
```

SERVIDOR.C

Aceptará peticiones de sus clientes tanto en TCP como en UDP, registrará todas las peticiones en un fichero de "log" llamado "peticiones.log" en el que anotará:

- Fecha y hora del evento
- Descripción del evento:

- Comunicación realizada: nombre del host, dirección IP, protocolo de transporte, nº de puerto efímero del cliente
- Una línea por cada uno de los mensajes intercambiado. Indicando en las órdenes recibidas
- Orden recibida: nombre del host, dirección IP, protocolo de transporte, el puerto del cliente y la orden.
- Para las respuestas
- Respuesta enviada: nombre del host, dirección IP, protocolo de transporte, puerto del cliente y la respuesta mandada.
- Comunicación finalizada: nombre del host, dirección IP, protocolo de transporte, nº de puerto efímero del cliente

Se ejecutará como un “daemon”

Aspectos relevantes del código

Utilizamos dos funciones auxiliares para la facilitación del programa.

generarRespuesta recibe el mensaje recibido por el cliente y la respuesta que este la va a dar. Aquí realizamos las comprobaciones de si el mensaje enviado por el cliente se ajusta a las opciones predeterminadas (que sea una de las ordenes establecidas y que termine en \r\n). Hacemos uso de la otra función auxiliar **leerLineaAleatoria** que devolverá aleatoriamente una de las preguntas y respuestas que se encuentran en el archivo "preguntas.txt", al tener este el formato "pregunta|respuesta", lo dividimos y realizamos las comprobaciones.

```
void generarRespuesta(char *m, char *respuesta) {
    if (strcmp(m, "HOLA\r\n") == 0 && flagHola == 0) {
        lineaAleatoria = leerLineaAleatoria(nombreArchivo);
        lineaCopia = strdup(lineaAleatoria);
        token = strtok(lineaCopia, "|");
        pregunta = token;
        token = strtok(NULL, "|");
        numero = token;
        intentos = rand() % 45 + 5; //Entre 5 y 50
        sprintf(respuesta, "250 %s#%d\r\n", pregunta, intentos);
        flagHola = 1;
        return;
    } else if (strcmp(m, "ADIOS\r\n") == 0) {
        flagHola = 0;
        strcpy(respuesta, "221 Cerrando el servicio\r\n");
        return;
    } else if (flagHola == 0) {
        strcpy(respuesta, "500 Error de sintaxis\r\n");
        return;
    } else if (strcmp(m, "+\r\n") == 0 && flagHola == 1) {
        lineaAleatoria = leerLineaAleatoria(nombreArchivo);
        lineaAleatoria = leerLineaAleatoria(nombreArchivo);
        lineaCopia = strdup(lineaAleatoria);
        token = strtok(lineaCopia, "|");
        pregunta = token;
```

```
        token = strtok(NULL, "|");
        numero = token;
        intentos = rand() % 45 + 5;
        sprintf(respuesta, "250 %s#%d\r\n", pregunta, intentos);
        return;
    }
    char *campo = strtok(m, "\r\n");
    if ((strncmp(campo, "RESPUESTA ", 10) == 0) && flagHola == 1) {
        campo = strtok(campo, " ");
        campo = strtok(NULL, " ");
        char *endptr;
        int num = strtol(campo, &endptr, 10);
        if (*endptr != '\0') {
            strcpy(respuesta, "500 Error de sintaxis\r\n");
            return;
        } else {
            if (intentos == 0) {
                strcpy(respuesta, "375 FALLO\r\n");
                return;
            } else if (num < atoi(numero) && intentos > 0) {
                intentos--;
                sprintf(respuesta, "354 MAYOR#%d\r\n", intentos);
                return;
            } else if (num > atoi(numero) && intentos > 0) {
                intentos--;
                sprintf(respuesta, "354 MENOR#%d\r\n", intentos);
                return;
            } else if (num == atoi(numero) && intentos > 0) {
                strcpy(respuesta, "350 ACIERTO\r\n");
                return;
            }
        }
    } else {
        strcpy(respuesta, "500 Error de sintaxis\r\n");
        return;
    }
    strcpy(respuesta, "500 Error de sintaxis\r\n");
    return;
}
```

leerLineaAleatoria lee aleatoriamente una línea del archivo "preguntas.txt" y la retorna.

```
char *leerLineaAleatoria(const char *nombreArchivo) {
    FILE *archivo = fopen(nombreArchivo, "r");
    if (archivo == NULL) {
        perror("Error al abrir el archivo");
        return NULL;
```

```
    }

    // Contar las líneas en el archivo
    int numLineas = 0;
    char c;
    while ((c = fgetc(archivo)) != EOF) {
        if (c == '\n') {
            numLineas++;
        }
    }

    // Generar un número aleatorio entre 1 y el número de líneas
    int numAleatorio = rand() % numLineas + 1;

    // Volver al principio del archivo
    rewind(archivo);

    // Leer la línea aleatoria
    char *linea = NULL;
    size_t len = 0;
    ssize_t read;
    int lineaActual = 0;

    while ((read = getline(&linea, &len, archivo)) != -1) {
        lineaActual++;
        if (lineaActual == numAleatorio) {
            // Quitar el salto de línea al final de la línea
            linea[strcspn(linea, "\n")] = '\0';
            break;
        }
    }

    fclose(archivo);
    return linea;
}
```

main generamos las variables necesarias para el establecimiento de la conexión y para la futura escritura de datos en el fichero "peticiones.log"

```
    srand(time(NULL));
    fd_set readmusk;
    int s_mayor;
    struct sockaddr_in servaddr, clientnaddr;
    socklen_t tamDir = sizeof(struct sockaddr_in);
    char respuesta[MAX], solicitud[MAX];
    char IPLocal[20], IPRemota[20];
```

```
    archivo = fopen("peticiones.log", "a");
    if (archivo == NULL) {
        perror("Error al abrir el archivo");
        return 1;
    }
    //Fecha y hora actual
    time_t tiempo_actual;
    struct tm *info_tiempo;
    char buffer[80];

    time(&tiempo_actual);
    info_tiempo = localtime(&tiempo_actual);

    //Dar valor al servaddr
    bzero(&servaddr, sizeof(servaddr));
    bzero(&clientnaddr, sizeof(clientnaddr));

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(PORT);

    //Crear socket sTCP --> socket
    sTCP = socket(AF_INET, SOCK_STREAM, 0);
    if(sTCP == -1) {
        perror("Error al crear el socket TCP");
        return 1;
    }

    //Asociar sTCP a la dirLocal --> bind
    if(bind(sTCP, (struct sockaddr *)&servaddr, sizeof(struct sockaddr_in)) ==
-1) {
        perror("Error al hacer bind");
        close(sTCP);
        return 1;
    }

    // Formatear la fecha y hora
    strftime(buffer, sizeof(buffer), "%d/%m/%Y\t%H:%M:%S", info_tiempo);

    //Poner el sTCP en modo escucha --> listen
    if(listen(sTCP, 5) == -1) {
        perror("Error al poner el socket TCP en modo escucha");
        close(sTCP);
        return 1;
    }

    //Crear socket sUDP --> socket
    sUDP = socket(AF_INET, SOCK_DGRAM, 0);
    if(sUDP == -1) {
        perror("Error al crear el socket UDP");
        return 1;
    }
    //Asociar sUDP a la dirLocal --> bind
```

```
    if(bind(sUDP, (struct sockaddr *)&servaddr, sizeof(struct sockaddr_in)) ==
-1) {
        perror("Error al hacer bind");
        close(sTCP);
        close(sUDP);
        return 1;
    }
    //Configurar señales
    signal(SIGTERM, handler);

    s_mayor = sTCP;
    if(sUDP > s_mayor) {
        s_mayor = sUDP;
    }

    setpgrp();
```

Generamos un proceso hijo con fork para la utilización del servidor en modo "daemon", en el caso -1, será un error, en el caso default cerramos todos los sockets y el archivo de escritura ya que no nos interesa ese proceso, y en el caso 0, eliminaremos los procesos de lectura y escritura de la terminal para que el servidor no bloquee a la hora de ejecutar el .sh, a demás comprobaremos si se está ejecutando en TCP o UDP.

```
switch (fork()) {
    case -1:
        //ERROR
        return 1;
    break;

    case 0:
        close(0);
        close(1);
        close(2);
        while(1) {
        FD_ZERO(&readmusk);
        FD_SET(sTCP, &readmusk);
        FD_SET(sUDP, &readmusk);
        if(select(s_mayor + 1, &readmusk, NULL, NULL, NULL) == -1) {
            //Error
            perror("Error en el select");
            return 1;
        }
        strcpy(IPLocal, inet_ntoa(servaddr.sin_addr));
        if(FD_ISSET(sTCP, &readmusk)) {
            //TCP (código a continuación)
        }

        if(FD_ISSET(sUDP, &readmusk)) {
```

```
            //UDP (código a continuación)
        }
    }
    break;

    default:
        close(sUDP);
        close(sTCP);
        close(s_mayor);
        fclose(archivo);
        return 0;
    break;
}
```

TCP

Si el servidor recibe a un cliente TCP, acepta la solicitud y crea un proceso, retornará 1 en caso de error, cerrará los sockets en caso default y en el caso 0 entrará en un bucle infinito que es donde recibirá (recv) y enviará (send) los mensajes al cliente y escribirá estos en el fichero ".log", en el caso de recibir la orden ADIOS, saldrá del bucle, cerrando el proceso.

```
sNuevo = accept(sTCP,(struct sockaddr *) &clientnaddr, &tamDir);
switch(fork()) {
    case -1:
        //Error
        perror("Error al crear el proceso hijo");
        return 1;
    break;
    case 0:
        close(sTCP);
        int i = 1;
        strcpy(IPRemota, inet_ntoa(clientnaddr.sin_addr));
        fprintf(archivo, "\nConexion TCP realizada >> %s\n", buffer);
        fprintf(archivo, "Puerto local >> %d\nPuerto remoto >> %d\n",
ntohs(servaddr.sin_port), ntohs(clientnaddr.sin_port));
        fprintf(archivo, "IP Local >> %s\nIP Remota >> %s\n", IPLocal,
IPRemota);
        while (i) {
            if(recv(sNuevo, respuesta, MAX, 0) == -1) {
                perror("Error al recibir la respuesta");
                close(sNuevo);
                return 1;
            }

            printf("Respuesta: %s", respuesta);
            fprintf(archivo, "[C] %s", respuesta)
            generarRespuesta(respuesta, solicitud);
```

```
            fprintf(archivo, "[S] %s", solicitud)
            if(send(sNuevo, solicitud, strlen(solicitud), 0) == -1) {
                perror("Error al enviar la respuesta");
                close(sNuevo);
                return 1;
            }
            if (strcmp(solicitud, "221 Cerrando el servicio\r\n") == 0) {
                close(sNuevo);
                i = 0;
            }
        }
    break;
    default:
        close(sNuevo);
}
```

UDP

Si el servidor recibe a un cliente UDP, recibe el mensaje de conexión, genera otro socket y simula otra conexión con el mismo cliente para crear un proceso (Este comportamiento lo realizamos para que reciba los mensajes en orden al ejecutarse concurrentemente evitando el problema de que lleguen los mensajes desordenados). Retornará 1 en caso de error, cerrará los sockets en caso default y en el caso 0 entrará en un bucle infinito que es donde recibirá (recvfrom) y enviará (sendto) los mensajes al cliente y escribirá estos en el fichero ".log", en el caso de recibir la orden ADIOS, saldrá del bucle, cerrando el proceso.

```
if(recvfrom(sUDP, respuesta, (sizeof(char) * MAX), 0, (struct sockaddr
*)&clientnaddr, &tamDir) == -1) {
    perror("Error al recibir la respuesta");
    close(sUDP);
    close(sNuevo);
    return 1;
}

sNuevo = socket(AF_INET, SOCK_DGRAM, 0);
if(sNuevo == -1) {
    perror("Error al crear el socket UDP");
    return 1;
}

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = INADDR_ANY;
servaddr.sin_port = htons(0);

if(bind(sNuevo, (struct sockaddr *)&servaddr, sizeof(struct sockaddr_in)) ==
-1) {
    perror("Error al hacer bind");
    close(sTCP);
    close(sUDP);
```

```
    return 1;
}


switch (fork()) {
    case -1:
        //Error
        perror("Error al crear el proceso hijo");
        return 1;
    break;

    case 0:
        strcpy(IPRemota, inet_ntoa(clientnaddr.sin_addr));
        close(sUDP);
        int i = 1;
        fprintf(archivo, "\nConexion UDP realizada >> %s\n", buffer);
        fprintf(archivo, "Puerto local >> %d\nPuerto remoto >> %d\n",
ntohs(servaddr.sin_port), ntohs(clientnaddr.sin_port));
        fprintf(archivo, "IP Local >> %s\nIP Remota >> %s\n", IPLocal,
IPRemota);

        fprintf(archivo, "[C] %s", respuesta);
        generarRespuesta(respuesta, solicitud);
        fprintf(archivo, "[S] %s", solicitud);
        if(sendto(sNuevo, solicitud, (sizeof(char) * MAX), 0, (struct sockaddr
*)&clientnaddr, sizeof(clientnaddr)) == -1) {
            perror("Error al enviar la solicitud");
            close(sNuevo);
            return 1;
        }
        if (strcmp(solicitud, "221 Cerrando el servicio\r\n") == 0) {
            close(sNuevo);
        }
        while(i) {
            if(recvfrom(sNuevo, respuesta, (sizeof(char) * MAX), 0, (struct
sockaddr *)&clientnaddr, &tamDir) == -1) {
                perror("Error al recibir la respuesta");
                close(sNuevo);
                return 1;
            }
            printf("Respuesta: %s", respuesta);
            fprintf(archivo, "[C] %s", respuesta);

            generarRespuesta(respuesta, solicitud);
            fprintf(archivo, "[S] %s", solicitud);

            if(sendto(sNuevo, solicitud, (sizeof(char) * MAX), 0, (struct
sockaddr *)&clientnaddr, sizeof(clientnaddr)) == -1) {
                perror("Error al enviar la solicitud");
                close(sNuevo);
                return 1;
            }
            if (strcmp(solicitud, "221 Cerrando el servicio\r\n") == 0) {
```

```
                close(sNuevo);
                i = 0;
            }
        }
        break;

    default:
        close(sNuevo);
}
```

preguntas.txt

Es el archivo en el que se almacenan las preguntas y respuestas. Se encuentran con el delimitador "|" para luego poder separar la pregunta de la respuesta. Este contiene 96 preguntas

```
¿Cuánto es 2 + 2?|4
¿Cuántos continentes hay en el mundo?|7
¿Cuántos planetas hay en nuestro sistema solar?|8
¿Cuántos lados tiene un triángulo?|3
¿Cuántos minutos hay en una hora?|60
¿Cuántas patas tiene un insecto típico?|6
.......
```

Makefile

Este Makefile compila dos programas, cliente y servidor, usando el compilador de C (gcc). El target "all" compila ambos programas. El target "clean" elimina los ejecutables generados.

```
CC = gcc

all: cliente servidor

cliente: cliente.c
    $(CC) -o cliente cliente.c

servidor: servidor.c
    $(CC) -o servidor servidor.c

clean:
    rm -f cliente servidor
```

lanzaServidor.sh

Contiene las ordenes de ejecución del programa. El & indica la simultaneidad de los programas, servidor no lo tiene ya que este tiene que ejecutarse antes para que los clientes puedan establecer la conexión.

```
./servidor
./cliente nogal TCP ordenes.txt &
./cliente nogal TCP ordenes1.txt &
./cliente nogal TCP ordenes2.txt &
./cliente nogal UDP ordenes.txt &
./cliente nogal UDP ordenes1.txt &
./cliente nogal UDP ordenes2.txt &
```

PRUEBAS

peticiones.log

Archivo resultante al ejecutar el script con los archivos de ordenes proporcionados en clase.