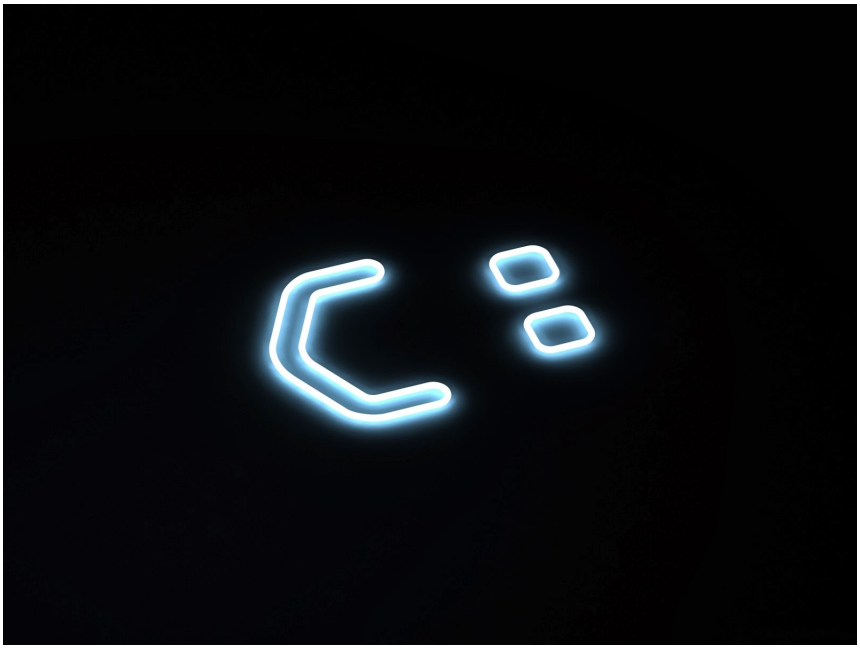
Pràctica Sistemes Operatius

Ariel Andreas Daniele - Clara Valls

arielandreas.daniele - clara.valls

09 - 01 - 2020



Índex

1 Disseny

Per començar a dissenyar la pràctica vam haver de fer un plantejament i una bona estructuració de tasques i més endavant de codi. Primer de tot vam crear un repositori a github, de manera que els dos membres del grup poguessin afegir fitxers i implementar codi sense preocupació de trepitjar el treball de l'altre i poder agafar la pràctica en el punt on l'havia deixat l'altre company. Per això també és important una divisió de tasques equitativa i eficient, tot i que aquest fet aniria en funció de la fase en la qual ens trobéssim, ja que en estar separada en 4 entregues, malgrat que no totes eren obligatòries, ens permetia poder repartir la feina progressivament. També serà important una bona modulació, ja que hi ha diverses funcionalitats i ens convenia separar el codi per arribar a la bona estructuració que plantejàvem al principi.

Pel que fa al disseny en si, aquest s'ha hagut de fer de manera progressiva en funció del temari que donàvem a classe i practicàvem en els laboratoris. Tot i que s'ha de tenir en compte el disseny o plantejament inicial, en l'apartat de recursos s'explica de manera detallada com s'ha dissenyat la pràctica i com ha sigut el seu disseny final. Vam començar amb els files decriptors, i per la primera fase ens va servir per acostumar-nos a emprar-los tant per l'escriptura i lectura, com per la utilització de fitxers, com el config.dat o el show_connections.sh més endavant.

Seguidament ens van presentar el Signals, i vam detectar ràpidament que per l'opció Exit era necessari reconfigurar el senyal SIGINT, és a dir, quan es prem Ctrl+C. D'aquesta manera podríem alliberar els recursos utilitzats i proporcionar una finalització correcta del programa. Els forks semblava que no tindrien molt de protagonisme, tot i que vam considerar que eren necessaris en la segona fase per l'opció Show Connections, ja que havíem d'executar l'script de les connexions i a la vegada llegir i mostrar el resultat que ens retornava, per això vam veure necessari duplicar el procés i així, amb ajuda d'una pipe, recollir les connexions disponibles per l'usuari. L'altre cas on emprem la funció fork és en realitzar el checksum a l'hora d'enviar àudios, on també necessitem un duplicament de processos i una pipe a cadascú per executar la funció md5sum i recollir el resultat. Com s'acaba d'esmentar, en els dos casos que fem forks és imprescindible l'ús de pipes, ja que es tracta de l'eina per comunicació de processos més útil, a més de que no és aplicable a processos que no s'hagin creat amb un fork. Per això els únics dos casos on ens ha fet falta emprar pipes és per la comunicació del procés pare i fill que apareixen en fer els dos forks.

Al tractar-se d'una pràctica enfocada al connexionat de servidors i clients, vam veure de seguida que aquesta s'havia de basar en sockets, i les funcionalitats a implementar es tractarien sobretot amb threads. Per tant, aquestes dues eines són segurament les més essencials del projecte, ja que són la base del principal objectiu del mateix. A més que els sockets són necessaris pel fet que

els programes han de poder executar-se i connectar-se des de diverses màquines. Per començar era necessari un servidor dedicat que escoltés possibles connexions de nous clients, la qual cosa també incita a utilitzar un thread per escoltar aquestes connexions.

Ja que la comunicació entre processos i fils d'execució sabíem que seria constant, caldria tenir un protocol de comunicació. Aquest se'ns facilitava a l'eStudy amb un annex, però igualment era necessari entendre'l i òbviament aplicar-lo correctament. En aquest protocol s'utilitzarà un únic tipus de trama, que podrà ser de dimensió variable i sempre estarà formada pels camps Type, Header, Lenght, Data. Tot i que només hi ha un tipus de trama, amb el Type y el Header podem controlar la informació que es transmet. Entre les trames més importants podem destacar la d'una nova connexió, la de Say per enviar missatges a un procés o amb l'opció Broadcast, la de show i download audios i la trama per sortir del programa (Exit). Tot l'intercanvi de dades i els avisos entre processos i fils d'execució es basarà en aquest protocol, per tant és una part força important del projecte.

L'última eina de les que ens van explicar que vam implementar a la pràctica són els semàfors. Tot i que sabiem de la utilitat per a temes de sincronització, només vam considerar els casos en els quals necessitàvem exclusió mútua. A més, com els casos on hi havia codi amb recursos compartits potencialment perillós es trobaven sempre en funcions de threads, vam aprofitar el mateix semàfor que ens facilita la llibreria pthread.h. A més aquesta eina era força simple d'emprar, ja que la inicialització era amb una constant, i la destrucció i les funcionalitats de wait i signal eren simplement tres funcions (lock, unlock i destroy). Per tant, només calia detectar la part crítica del codi i encapsular-la entre un lock i un unlock i d'aquesta manera el recurs ja estaria protegit i l'exclusió mútua aplicada.

1.1 Diagrama d'activitats

A continuació explicarem el diagrama d'activitats del nostre programa. Tindrem en compte una execució lògica, és a dir, on ens connectarem a cap port sense haver mirat quins estan disponibles, no descarregarem un àudio o enviarem missatges a un usuari sense estar connnectats, etc. Considerem també que l'usuari no escollirà una opció incorrecta ni buscarà àudios o connexions (usuaris) inexistents.

Evidentment el diagrama tindrà el seu inici quan s'executi el programa, passant com a únic argument el fitxer de configuració. El primer que es farà és llegir i extreure la informació necessària d'aquest fitxer. A continuació es crearà el servidor i s'iniciarà el thread Escolta per escoltar noves connexions de clients. Seguidament es mostrarà un prompt a l'usuari el qual apareixerà cada cop que

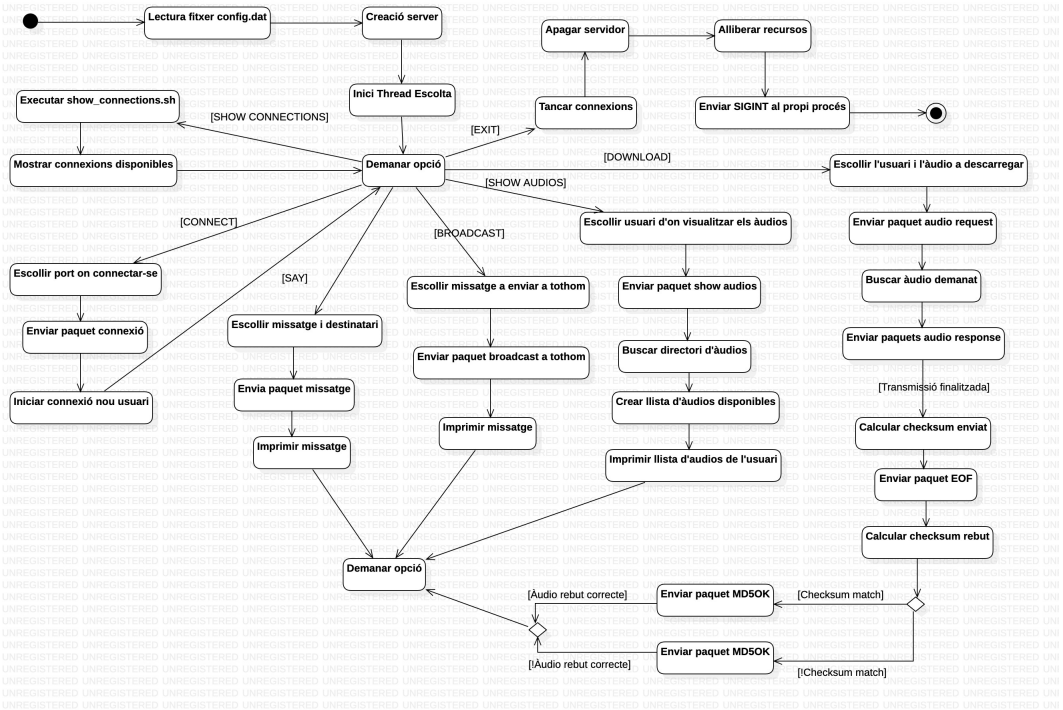
finalitzi una funcionalitat i l'usuari vulgui seleccionar una opció. En total podrà escollir entre 7 opcions.

La primera opció es Show Connections, que necessitarà executar l'script show_connections.sh per tal de poder mostrar a l'usuari els ports on es pot connectar. L'opció Connect haurà d'anar acompanyada del port on ens volem connectar, el qual segurament hàgem detectat en executar la primera opció. Una vegada tinguem una o més connexions actives, podrem interactuar amb la resta d'usuaris. Podem enviar un missatge a un usuari en concret amb l'opció Say, o enviar un missatge de difusió a tothom amb els quals estiguem connectats amb l'opció Broadcast. En els dos casos haurem d'enviar cada cop el paquet amb el missatge a l'usuari en qüestió, i aquest el rebrà i el visualitzarà per pantalla.

L'opció Show Audios haurà d'anar acompanyada de l'usuari del qual vulguem veure els àudios que té a la seva respectiva carpeta. Haurem de buscar el directori de l'usuari i crear una llista de tot el seu contingut, que posteriorment mostrarem per pantalla. L'opció download és segurament la més complexa. Per començar haurem d'especificar tant l'usuari com l'àudio en concret que vulguem descarregar. Primer enviarem un paquet del tipus audio request i buscarem l'àudio a descarregar. Una vegada trobat anirem transmitent el contingut de l'àudio amb paquets del tipus audio response, els quals seran tots de la mateixa mida i s'aniran enviant d'un a un per poder escriure les dades del paquet al nou fitxer d'àudio que apareixerà al finalitzar la descàrrega. En acabar la transmissió, calcularem el checksum enviat per veure si s'ha transmès correctament, i s'enviarà el paquet EOF (End of File). Seguidament calcularem el checksum rebut i segons si coincideix amb el calculat anteriorment, sabrem si l'arxiu que hem descarregat és el que volíem o on coincideix amb la descàrrega desitjada.

Finalment tindrem l'opció Exit, a la qual accedirem si escrivim exit en el prompt o si premem Ctrl+C, per tant hem de considerar els dos casos. Abans de sortir definitivament del programa, haurem de tancar les connexions amb els clients, apagar el servidor, alliberar tots els recursos emprats i enviar-nos a nosaltres mateixos un SIGINT per finalitzar l'execució del programa.

Cal mencionar que a l'entrega, inclourem el diagrama d'activitats en format jpg, per si no s'aprecia suficientment a la memòria i es vol observar més detingudament.



1.2 Estructures de dades

L'elecció i la creació de les estructures de dades no ha significat un problema. A més dels tipus primitius coneguts per tots, hem emprat alguns tipus específics de llibreries i ens hem creat els nostres propis structs.

Amb la llibreria pthread.h hem pogut emprar 3 tipus especials en la definició de variables de la pràctica, els quals han sigut força útils. Per començar en la primera línia del codi següent veiem el *pid_t*, el qual referencia l'ID del procés, és a dir, del thread. Aquest l'hem emprat a l'hora de fer el fork i per després poder distingir si es tractava del procés fill o pare. En la segona tenim el tipus propi del thread, que en el nostre cas el necessitem pel thread que escolta les noves connexions. Finalment, aquesta llibreria també incorpora un semàfor d'exclusió mútua propi de threads, el qual ens ha sigut de molta utilitat en la gestió dels arrays de threads.

```
pid_t pid = fork ();

pthread_t escolta;

pthread_mutex_t sWrite = PTHREAD_MUTEX_INITIALIZER;
```

A més d'aquests també hem emprat el tipus ssize_t, que ens ha servit a l'hora de llegir fitxers, comandes o intercanvis d'informació entre processos. També hem emprat l'struct sockaddr_in, que ens era imprescindible per la creació del memset dels sockets. Per acabar, recalcar que hem fet ús de l'eina extern, que serveix per poder agafar el valor d'una variable externa d'un altre mòdul, evitant així haver d'implementar un getter i un setter.

```
ssize_t nbytes;         //guardarà el número de bytes que ha llegit

struct sockaddr_in s_addr;

extern Config config;   //valors del fitxer de configuració
```

Com a primera estructura de dades complexa diferent dels tipus que hem emprat han sigut els arrays. Hem decidit no emprar llistes, sobretot pel fet que hem prioritzat l'ús de la memòria dinàmica i fer les estructures lineals amb arrays en lloc d'amb llistes facilita molt la implementació. D'aquesta manera, la majoria d'arrays que tractem seran dinàmics. En el cas de les cadenes de caràcters també hem evitat fer-les estàtiques, per tant hi ha hagut un gran ús de punters a caràcters, els quals hem hagut de gestionar la memòria amb mallocs, reallocs, asprintf i altres comandes, com la resta d'arrays dinàmics. A més dels caràcters també hem fet arrays dinàmics d'altres tipus primitius i fins i tot de structs propis, com és el cas dels clients connectats. Veiem que la declaració és com un punter normal i que per crear memòria és necessari un simple malloc. Al mateix temps, si volem redimensionar l'array amb un realloc és força senzill. D'aquesta manera, els arrays dinàmics han sigut una eina molt recurrent durant la pràctica i que a base de repetició se'ns ha fet molt fàcil gestionar.

```
Conn_cli *conn_clients;     //els que s'han conectat

if(qClients == 0){
    //creem l'array on guardarem els clients que s'han connectat
    conn_clients = (Conn_cli*)malloc(sizeof(Conn_cli));
}
else{
    //ampliem l'espai de memòria de l'array
    conn_clients = (Conn_cli*)realloc(conn_clients, sizeof(Conn_cli)*
    (qClients + 1));

}
```

Tot i que ja sabem com funciona un array estàtic en C i que nosaltres no hem emprat molts, podem destacar alguns del nostre codi també per exemplificar en quins casos hem considerat oportú fer-ne ús. Un exemple és els dos files descriptors que conformen una pipe, un d'escriptura i un de lectura. En aquest

cas està clar que necessitem un array de dos enters. Un altre cas que ens ha semblat bastant útil és l'array de cadenes de caràcters per tal de tenir els arguments que li passarem al procés que executarà l'script show_connections.sh.

```
int fd[2];          //variable que farà de pipe

//creem els arguments a passar-li al procés show_connections.sh
char * argv[5] = {"./show_connections_v2.sh", sysports[0],
sysports[1], ip, NULL};
```

Pel que respecta als structs que ens hem creat nosaltres, han sigut un total de 5. El primer que vam necessitar va ser el del fitxer config. Com que la lectura d'aquest fitxer era essencial pel desenvolupament de la pràctica vam decidir fer un senzill struct amb tota la informació del mateix.

```
typedef struct{
    char *user;
    char *dirAudios;
    char *ip;
    int port;
    char *ipWeb;
    char **sysports;
    int sockfd;
}Config;
```

Per l'enviament de paquets necessitàvem un protocol de comunicació, i aquest es tradueix en un struct amb els 4 elements que té qualsevol paquet.

```
typedef struct{
    unsigned char type;
    char * header;
    short length;
    char * data;
}Protocol;
```

Finalment, per tal de simplificar-nos la vida, ja que vam tenir problemes amb la gestió d'usuaris, vam decidir fer un struct per cada llista de clients i servidors amb la informació que necessitàvem. Del servidor ens interessava emmagatzemar el port al qual ens connectaven els clients, el file descriptor del socket i l'usuari. Mentre que pels clients no necessitàvem el port. També requeríem d'un thread en el servidor dedicat que escoltés les connexions, per tant per simplificar-nos la vida vam crear un struct (UserThread) amb el propi thread, l'usuari i un listener on guardarem el file descriptor del socket.

```
typedef struct {
    int port;
    int sockfd;
    char *user;
}Conn_serv;

typedef struct {
    int sockfd;
    char *user;
}Conn_cli;

typedef struct{
    pthread_t t;
    char *user;
    int listener;
}UserThread;
```

Hem de destacar també l'ús dels defines en tots els mòduls de la pràctica. Per exemple, la definició de totes les opcions de la pràctica com un número. Sense obviar els defines de cadenes de caràcters que ens simplificàvem molta tecla amb les escriptures per pantalla i l'intercanvi d'informació.

```
#define SHOW_CONNECTIONS 1
#define CONNECT 2
#define SAY 3
#define BROADCAST 4
#define SHOW_AUDIOS 5
#define DOWNLOAD 6
#define EXIT 7


#define TRANS_END "\n[%s] %s correctly downloaded"
#define DOWNLOADING "\nDownloading...\n"
#define MD5KO "Audio checksum doesn't match\n"
#define AUDIOKO "This audio file does not exists\n"
#define COOL "\n[%s] Cool!\n"
#define SHOW_REBUT "Audios list solicited\n"
```

1.3 Recursos del sistema utilitzats

1.3.1 File descriptors

Un file descriptor és una clau a una estructura de dades resident en el nucli, que conté detalls de tots els fitxers oberts. En programació en C s'interpreta amb el tipus enter i es tractarà d'un element bàsic per a crides al sistema d'entrada sortida.

Els tres file descriptor estàndard són el 0 (entrada), l'1 (sortida) i el 2 (error), tot i que una variable d'aquest tipus pot prendre qualsevol valor enter. Les funcions que se'ns permet utilitzar amb aquest recurs són open(), close(), write() i read().

Un exemple clar de file descriptor que emprem a la pràctica, a més de la lectura per teclat i l'escriptura per pantalla, és el cas de la lectura del fitxer config. Primer de tot declarem el file descriptor com un enter. Amb la funció open obrim el fitxer i se li atorga un valor a l'enter f. Si aquest és menor a 0, hi ha hagut algun error, en cas contrari continuem amb l'execució. Amb la funció readUntil anirem llegint el fitxer línia per línia i podrem omplir la nostra variable config amb els camps necessaris del fitxer. Finalment hem d'alliberar aquest recurs, i el tanquem amb un simple close(f).

```
Config lecturaFitxer(const char *fitxer){
    int f;                  //file descriptor del fitxer
    char * aux;             //cadena auxiliar per llegir valors enters

    //obrim el fitxer
    f = open(fitxer, O_RDONLY);
    if (f < 0)
    {
        write(1, ERR_FILE, strlen(ERR_FILE));
        config.user = NULL;
    }

    config.user = readUntil(f, '\n', '\0');
    config.dirAudios = readUntil(f, '\n', '\0');
    config.ip = readUntil(f, '\n', '\0');

    aux = readUntil(f, '\n', '\0');
    config.port = atoi(aux);
    free(aux);
    .
    .
    .
    close(f);
    return config;
```

```
}
```

1.3.2 Signals

Un signal és un mecanisme del nucli per comunicar events als processos. El sistema operatiu s'encarrega que el procés que rep el senyal la tracti immediatament. Així, cada signal té associat un nom i un número. Hi ha molts signals amb un valor i funció per defecte, com per exemple el SIGINT, que es llença al prémer Ctrl+C durant l'execució d'un programa, finalitzant el mateix. Però, podem canviar el comportament d'un signal reemplaçant les funcions que compleixen per defecte per una funció pròpia del codi, per poder capturar senyals i tractar-les. Això sí, és convenient en alguns casos que després de capturar el senyal donar-li el seu valor per defecte novament.

Pel que fa al nostre programa, només emprem un signal, en aquest cas el SIGINT, que es llença quan durant l'execució del programa premem Ctrl+C. El que hem fet ha sigut, en el main, reconfigurar el senyal amb la funció signal, fent que enlloc de finalitzar el programa vagi a la funció stopAll.

```
//reconfigurem la senyal SIGINT
signal(SIGINT, stopAll);
```

Un cop s'executa la funció stopAll, farem dues coses. La primera serà cridar la funció optionExit on es farà el necessari abans de finalitzar el programa, és a dir, tancar les connexions, apagar el servidor, alliberar els recursos, etc. Aquesta funció per acabar li retornarà el valor per defecte que tenia el signal SIGINT. Per tant, quan s'executi la línia raise(SIGINT), es llançarà el senyal SIGINT cap al nostre programa, i com l'hem reconfigurat a com estava per defecte, el programa finalitzarà satisfactòriament.

```
void stopAll(){
    optionExit();
    raise(SIGINT);
}

void optionExit(){
    write(1,DISCONNECT, strlen(DISCONNECT));

    //avisem als clients
    tancaConnexions();

    //alliberem la memòria de les connexions
    freeConnections();

    //deixem d'escoltar connexions
    apagaServidor();
```

```
    //reconfigurem signals
    signal(SIGINT, SIG_DFL);
}
```

1.3.3 Forks

Un fork és una funció per clonar o duplicar procés. Sabem que un procés és un programa en execució, el qual tindrà un PID o identificador de procés. Així, si fem un fork en el codi d'un programa, es duplicarà i es crearà un "fill" que continuarà a executar-se just seguidament la línia on s'ha fet el duplicat.

Per poder controlar la jerarquia de processos, serà recomanable just després d'un fork fer un codi condicional, en funció de si es tracta del procés fill (la funció retornarà un 0), el procés pare (se li retornarà el pid del fill, per tant sempre major a 0) o si hi ha hagut algun error, cas en el qual el fork retornaria un -1.

El nostre programa requereix aquest recurs en dos ocasions, i en els dos casos es combinarà amb una pipe, que explicarem en el següent apartat. La primera és necessària en l'opció Show Connections. En aquesta opció realitzarem un fork de la variable pid, i justament després hi haurà un switch amb els tres casos que es poden donar, que sigui el fill (pid=0), que hi hagi algun error (pid=-1) i en la resta de casos significaria que és el pare, ja que el pid serà major a 0.

```
pid_t pid = fork ();
switch (pid){
    case 0:
        close(fd[0]);
        dup2(fd[1], 1);

        //executem show_connections.sh
        if(execvp(argv[0], argv) < 0){
            write(1, ERR_CONN, strlen(ERR_CONN));
        }
        break;
    case -1:
        write(1, ERR_CONN, strlen(ERR_CONN));
        break;
    default:
        close(fd[1]);
        //esperem que acabi d'escriure
        wait(NULL);
        //llegim del pipe i busquem els ports
        buscaPorts(fd[0], myPort);
        close(fd[0]);
```

```
        break;
}
```

En el cas del fill primer tancarem el file descriptor de la pipe que no ens interessa, en aquest cas el de lectura, executarem el script show_connections.sh per a mostrar les connexions disponibles per pantalla. En el cas del pare, una vegada tanquem el file descpritor d'escriptura de la pipe, esperarem a que acabi d'escriure. Seguidament llegirem el que ens envia la pipe i buscarem els ports disponibles que mostrarem a l'usuari per pantalla.

El segon cas és molt similar, per tant s'explicarà en l'apartat de Pipes per donar-li èmfasi a aquell recurs del sistema i no ser repetitiu amb els forks.

1.3.4 Pipes

Aquest és el primer mecanisme de comunicació de processos que va aparèixer. Està format per 2 file descriptors, un serveix per llegir i l'altre per escriure. Tracten un conjunt de bytes sense format (màxim 64 KB), realitzant lectures destructives i bloquejants. En ser unidireccionals, és important tancar el file descriptor que no emprem en la pipe, ja que ens suposaria un malbaratament de recurs. De la mateixa manera, si volguéssim una comunicació bidireccional entre processos, hauríem de crear dos a cadascú i tancar en cada cas el file descriptor contrari a l'altre pipe amb el qual ens comuniquem.

Com hem dit en l'apartat anterior, hi havia un segon cas en el qual necessitàvem emprar pipes, sempre acompanyat d'un fork, ja que si no tindria poc sentit. En aquest cas realitzem el Checksum per verificar que els paquets de l'àudio que s'envien arribem correctament i la informació coincideix. Primer declarem la pipe com un array de 2 file descriptor. Amb la funció pipe(fd) la creem, i en cas d'error notifiquem immediatament i marxem de la funció. Seguidament fem el fork de la variable pid. Un altre cop el procés es comportarà diferent en funció de si és el fill (0), un error (-1) o el pare(¿0).

```
char * calculaChecksum (char *path){
    char *checksum;
    int fd[2];
    char * argv[3] = {"md5sum", path, NULL};

    if (pipe(fd) == -1){
            write(1, ERR_PIPE, strlen (ERR_PIPE));
            exit(-1);
    }

    pid_t pid = fork ();
    switch (pid){
        case 0: //fill
```

```
            close(fd[0]);
            dup2(fd[1], 1);
            //executem md5sum
            execvp(argv[0], argv);
            break;
        case -1:
            write(1, ERR_CONN, strlen(ERR_CONN));
            break;
        default: //pare
            close(fd[1]);
            //esperem que acabi d'escriure
            wait(NULL);
            //llegim del checksum i enviem
            checksum = readUntil(fd[0], ' ', '\0');
            close(fd[0]);
            break;
    }
    return checksum;
}
```

En el cas del fill primer tancarem el file descriptor de lectura de la pipe. Després dupliquem el file descriptor d'escriptura amb la funció dup2 i finalment executem el md5sum. El pare en canvi haurà de tancar el file descriptor d'escriptura i esperar que la pipe acabi d'escriure. A continuació llegirà el checksum que li arriba amb la funció readUntil i l'envia. Per acabar es tanca també el file descriptor de lectura per no malbaratar recursos

1.3.5 Sockets

Un socket és una eina de comunicació que ens permetrà intercanviar dades entre dos o més ordinadors. Aquestes comunicacions entre servidor i client pot produir-se de forma remota o local, per tant podem connectar dos ordinadors encara que estiguin físicament lluny. Per a aconseguir això es configura l'asociació a partir d'una adreça IP i un port. Cal destacar que realment els sockets els emprarem com un file descriptor, per tant seran simplement enters.

Podem diferenciar dos tipus de sockets, segons si són orientats o no orientats a connexió. Els orientats a connexió (TCP) probablement tinguin un intercanvi de dades més lent que el no orientat a connexió (UDP) però ens assegurem que tots els paquets arribin al destí.

Pel que respecta a la nostra pràctica, tota la comunicació entre el servidor Trinity i els clients es basa en sockets i threads.

En el següent codi podem observar com realitza la connexió el socket servidor de l'aplicació. Rebem per paràmetre la IP i el port on es connectarà el servidor i amb la funció socket creem el sockfd (file descriptor) del servidor.

```
int connectServer(const char* ip, int port){
    //iniciem la connexió del servidor
    int sockfd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
    struct sockaddr_in s_addr;

    memset (&s_addr, 0, sizeof (s_addr));
    s_addr.sin_family = AF_INET;
    s_addr.sin_port = htons (port);
    if (inet_aton (ip, &s_addr.sin_addr) == 0) {
        return -1;
    }

    if (bind (sockfd, (void *) &s_addr, sizeof (s_addr)) < 0) {
        return -1;
    }

    listen (sockfd, 3);

    qClients = 0;
    qServ = 0;

    return sockfd;
}
```

1.3.6 Threads

Els threads són mecanismes que generen fils d'execució, permetent realitzar una sola tasca per fil i així augmentar l'eficiència del programa. Sempre hi ha un thread pare que crea a la resta, però en compartir molts recursos hem d'anar amb cura a l'hora de programar. Això si, sabem que si el pare mor la resta de threads creats per ell es destrueixen.

Pel que fa al nostre codi, havíem d'implementar el thread del servidor dedicat que escolta contínuament les noves connexions dels clients. Aquest està codificat amb la funció threadEscolta, la qual rep com a paràmetre la configuració. Mentre el socket del servidor segueixi encès esperarem noves connexions, mentre que si el newsock ens dóna negatiu apagarem el servidor i deixarem d'escoltar possibles connexions.

```
static void *threadEscolta (void *config){
    int newsock;                    //socket que vol connectar-se
    char *clientName;               //nom del client que es connecta
```

```
    Config *c = (Config *) config;  //fitxer de configuració
    Protocol p;                     //protocol de comunicació

    struct sockaddr_in c_addr;
    socklen_t c_len = sizeof (c_addr);
    mySockfd = c->sockfd;

    while (apaga){
        newsock = accept (mySockfd, (void *) &c_addr, &c_len);
        if (newsock < 0) {
            apaga = 0;
        }
        else{
            //esperem el missatge del client que s'ha connectat
            p = llegeixPaquet(newsock);

            if (p.data == NULL){
                //És una connexió falsa. La tanquem
                close(newsock);
            }
            else {
                asprintf(&clientName, "%s", p.data);
                afegeixClient(newsock, c->user, clientName);
                alliberaPaquet(p);
            }
        }
    }
    return NULL;
}
```

Perquè hi hagi una nova connexió haurem d'iniciar el thread client, la qual cosa es farà en la següent funció. Aquest rep el thread del client i el nom del nou usuari. Tota la inicialització l'encapsularem en un semàfor d'exclusió mútua (pthread_mutex) i paral·lelament haurem de redimensionar l'array de clients. Una vegada el nou client tingui actualitzat el seu sockfd i el nom d'usuari el crearem amb la funció que ens facilita la llibreria pthread.h (pthread_create).

```
void iniciaThreadClient(Conn_cli* client, char *user){
    pthread_mutex_lock(&mtxqC);
    //augmentem la mida de l'array
    if(qTCli == 0){
        tCli = (UserThread*)malloc(sizeof(UserThread));
    }
    else{
        tCli = (UserThread*)realloc(tCli, sizeof(UserThread)*
        (qTCli+1));
```

```
    }

    tCli[qTCli].user = user;
    tCli[qTCli].listener = client->sockfd;

    pthread_create(&(tCli[qTCli].t), NULL, threadCli, client);
    qTCli++;
    pthread_mutex_unlock(&mtxqC);
}
```

Una vegada tinguem connectats el nou client amb el servidor l'usuari podrà començar a fer ús de les funcionalitats que presenta la pràctica. La majoria d'aquestes funcions les controlarem enviant paquets d'informació entre threads, la qual cosa es gestiona a les funcions threadServ i threadCli. Veiem una exemplificació del codi del thread del client. En un while infinit que només ometrem quan ja no estigui connectat el client, llegirem els paquets que ens arriben un a un i segons el tipus de paquet farem una cosa o un altra. Per exemple si ens arriba un missatge d'un altre client amb l'opció Say (case 0x02), imprimirem aquesta informació per pantalla degudament i tornarem a mostrar el prompt per defecte de cada client.

```
static void *threadCli (void *client){
    Conn_cli *aux= (Conn_cli *) client;
    Conn_cli *c;                            //info del nou client
    Protocol p;                             //protocol de comunicació
    char connectatC = 1;                    / quan aturar el thread
    char *audiosShow;
    char *missatge;

    c = (Conn_cli*)malloc(sizeof(Conn_cli));
    c->user = (char*)malloc(sizeof(char)*strlen(aux->user) + 1);
    strcpy(c->user, aux->user);
    c->sockfd = aux->sockfd;

    while (connectatC){
        //escoltem si el client ens envia un missatge
        p = llegeixPaquet(c->sockfd);

        switch(p.type){
            case 0x02:
                if(strcmp(p.header, "[MSG]") == 0){
                    imprimeixMissatge(p.data, c->user);
                    imprimeixPrompt();
                }
                break;
```

```
            case 0x03:
                if(strcmp(p.header, "[BROADCAST]") == 0){
                    imprimeixMissatge(p.data, c->user);
                    imprimeixPrompt();
                    enviaPaquet(c->sockfd, 0x03, "[MSGOK]", 0, NULL);
                }
                break;

            case 0x04:
                if(strcmp(p.header, "[SHOW_AUDIOS]") == 0){
                    enviaPaquet(c->sockfd, 0x02, "[MSG]",
                    strlen(SHOW_REBUT), SHOW_REBUT);
                    free(missatge);
                    audiosShow = buscaAudios();
                    enviaPaquet(c->sockfd, 0x04, "[LIST_AUDIOS]",
                    strlen(audiosShow), audiosShow);
                    free(audiosShow);
                }
                break;
            .
            .
            .
        }

        alliberaPaquet(p);
    }
    return NULL;
}
```

1.3.7 Semàfors

Un semàfor dóna accés al recurs a un dels processos i l'hi nega als altres mentre el primer no acabi de manipular aquest recurs. El seu funcionament és similar al que seria una variable comptador. Sempre que el comptador del semàfor sigui major o igual a 0 se li permetrà l'accés al recurs i seguidament disminuirà aquest comptador. Si seguidament volem accedir a aquest recurs i el comptador és negatiu, aquest quedarà bloquejat i no continuarà amb l'execució fins que incrementi el semàfor.

Per a utilitzar aquest recurs, a més de la òbvia inicialització i destrucció, només tenim dues funcions. La funció wait() disminueix el comptador, per tant només en el cas que després de fer un wait el comptador del semàfor sigui positiu continuarà l'execució i podrem accedir al recurs. Si no és així quedarà bloquejat fins que es cridi a l'altra funció, és a dir, fins que es faci un signal() al mateix semàfor. La funció signal incrementa el comptador del semàfor.

Si utilitzem un semàfor per a exclusió mútua, per començar haurem d'inicialitzar-lo a 1, i cada vegada que vulguem modificar un recurs compartit, haurem d'encapsular aquella part de codi amb un wait() al principi i un signal() al final. D'aquesta manera evitarem que es toqui la mateixa variable simultàniament, ja que després de fer el wait el recurs quedarà bloquejat per a la resta de processos o fils d'execució fins que es faci el signal corresponent i el recurs torni a estar disponible.

Si en canvi volguéssim utilitzar un semàfor per a sincronització, les funcions a emprar són les mateixes amb un criteri diferent, començant per inicialitzar-lo a 0. Tindrem un tros de codi que ens interessarà no començar fins que no acabi un altre, per tant haurem d'incloure un wait al principi del codi que depèn de l'altre, el qual un cop s'executi completament hauria de tenir un signal per tal d'avisar al wait i que comenci a executar-se el codi que el segueix.

Pel que fa a la nostra pràctica, només ens ha fet falta emprar semàfors d'exclusió mútua, i per a aconseguir-ho hem decidit utilitzar el semàfor de threads pthread_mutex_t, el qual està inclòs en la llibreria pthread.h. Els casos en els quals hem hagut de fer exclusió mútua són tres. El primer de tots és imprescindible per la creació del l'inicialització dels threads, tant client com servidor. Aquest està exemplificat a l'apartat de threads i bàsicament es tracta d'un semàfor que encapsula tota l'inicialització d'un thread, evitant que més d'un thread s'inicialitzi a la vegada i procurar que aquesta acció es faci correctament.

El segon cas és per mostrar missatges per pantalla, ja que és un requeriment necessari que els missatges pel terminal de cada programa es facin seqüencialment, i evitar que dos processos vulguin escriure al mateix temps en el mateix file descriptor. Veiem amb el codi que tenim a continuació que la inicialització del semàfor es realitza al mateix temps que instanciam la variable, amb el define que ens facilita la llibreria. Després hem de detectar el tros de codi potencialment conflictiu i que volem protegir, en aquest cas es tracta de la funció write, per tant abans d'aquesta línia fiquem un mutex_lock, que seria l'equivalent a un wait(), i després del codi protegit fem un mutex_unclock, que fa la funcionalitat del signal(). Finalment haurem d'alliberar el recurs utilitzar amb la funció destroy de la mateixa llibreria.

```
pthread_mutex_t sWrite = PTHREAD_MUTEX_INITIALIZER;
.
.
.
pthread_mutex_lock(&sWrite);
write(1, missatge, strlen(missatge));
pthread_mutex_unlock(&sWrite);
.
```

```
.
.
pthread_mutex_destroy(&sWrite);
```

L'últim cas ens ve amb la necessitat d'actualitzar l'array de clients i servidors cada cop que es desconnecta un d'aquests. Com que les variables array que guarden la informació dels threads servidors i clients així com la quantitat de cadascú són globals, pot haver-hi conflictes i que dos processos o fils d'execució vulguin accedir paral·lelament a la mateixa variable.

A nivell de codi és molt similar al cas anterior, ja que tenim la instància i la inicialització a la mateixa línia. Per exemple, si ens fiquem en el cas que es desconnecta un servidor, haurem, després d'haver fet el join, fer un mutex_lock, és a dir un wait. Si ningú està emprant els recursos compartits, podrem avançar i fer els joins de l'array de servidors, que reordenarà l'estructura de dades shiftant les posicions necessàries, i el decrement de la quantitat de servidors (qTServ). Quan hàgem actualitzat la informació podrem enviar un signal i així avisar als altres processos que ja poden modificar aquest recurs si és que cal. Òbviament al final haurem d'alliberar el recurs emprat amb un mutex_destroy.

```
pthread_mutex_t mtxArrayS = PTHREAD_MUTEX_INITIALIZER;
.
.
.
pthread_join(tServ[i].t, NULL);
pthread_mutex_lock(&mtxArrayS);
shiftJoins(tServ, user, qTServ);
qTServ--;
pthread_mutex_unlock(&mtxArrayS);
.
.
.
pthread_mutex_destroy(&mtxArrayS);
```

1.3.8 Dup2 i exevcp

Com a eines externes a les que ens han ensenyat a classe de teoria, utilitzarem execvp i dup2 de forma conjunta. Necessitarem, a més, les eines fork i pipe per algunes funcionalitats, les quals ja hem explicat amb anterioritat.

L'eina execvp s'utilitza per executar una funció dins el programa actual. Un cop acaba, finalitza el programa. Necessita rebre un array amb el nom de la funció a executar a la primera casella, seguida dels paràmetres necessaris per a la seva execució i posant a NULL l'última casella de l'array. Per poder mantenir l'execució del progarama Trinity, ho utilitzarem conjuntament amb l'eina fork,

fent que el fill executi la funció i el pare segueixi amb l'execució del nostre programa.

L'eina dup2 s'utilitza per canviar el valor d'un file descriptor. En el nostre cas, s'utilitzarà aquesta comanda perquè la sortida de l'execució de la comanda execvp s'enviï per un pipe en comptes de mostrar-se pel terminal. És a dir, canviarem el file descrptor 1 pel del pipe corresponent.

Podem veure un exemple de codi, el qual ja hem vist anteriorment per explicar les pipes, que mostra l'ús de les quatre eines esmentades que funcionen conjuntament.

```
char * calculaChecksum (char *path){
    char *checksum;
    int fd[2];
    char * argv[3] = {"md5sum", path, NULL};

    if (pipe(fd) == -1){
            write(1, ERR_PIPE, strlen (ERR_PIPE));
            exit(-1);
    }

    pid_t pid = fork ();
    switch (pid){
        case 0: //fill
            close(fd[0]);
            dup2(fd[1], 1);
            //executem md5sum
            execvp(argv[0], argv);
            break;
        case -1:
            write(1, ERR_CONN, strlen(ERR_CONN));
            break;
        default: //pare
            close(fd[1]);
            //esperem que acabi d'escriure
            wait(NULL);
            //llegim del checksum i enviem
            checksum = readUntil(fd[0], ' ', '\0');
            close(fd[0]);
            break;
    }
    return checksum;
}
```

2 Problemes observats

Durant la realització d'aquesta pràctica ens hem trobat amb una gran quantitat de problemes, sobretot pel fet que l'hem anat desenvolupant durant tot el curs, la qualitat de codi i el poc marge d'error que se'ns demanava.

Un petit problema que al final ens vam adonar que no era tan greu vas ser la utilització del script show_connections.sh en funció de si utilitzàvem un mac o un ordinador windows, ja que cada membre de la parella en tenia un. El problema va sorgir quan estàvem implementant l'opció Show Connections, on vam veure que en executar l'script en mac apareixien unes línies que en teoria l'script mateix hauria d'evitar que sortissin. Però en executar-ho amb el mateix codi amb windows això no ocorria, i anava segons el previst. En aquest cas no afectava al funcionament de la pràctica, però en implementar l'opció Download ens vam adonar que, al tornar a utilitzar aquest script en aquesta opció el checksum no coincidia mai, mentre que amb el mateix codi a windows sí que funcionava. Per tant, al final vam ometre aquests petits detalls i vam seguir programant tenint en compte aquest fet en executar amb mac.

Durant la implementació de la funcionalitat Download vam veure alguns problemes, ja que l'arxiu d'àudio semblava que es descarregava correctament i fins i tot tenia la mateixa quantitat de bytes que l'original. Malgrat això, a l'hora d'escoltar-lo aquest no tenia els mateixos minuts i en alguns moments no s'escoltava igual, per tant vam detectar que la descarrega no s'havia fet satisfactòriament. Al final, amb l'ajuda del checksum ens vam adonar que sempre s'enviaven els paquets però la informació d'aquest no era correcta. Això era degut al fet que en fer el read per llegir els paquets se'ns afegia un '\0', i per tant sempre hi havia informació que es perdia.

A la mateixa opció vam tenir un altre problema amb el '\0', i és que en utilitzar la funció strlen ens detectava els '\0' com el seu valor en ascii, és a dir com un 10. Per tant, sempre que trobava un '\0' deixava de llegir la informació i el paquet rebut no coincidia amb l'enviat. Així hem après que la funció strlen no és recomanable per l'enviament de paquets. Després d'algunes hores de debuggar el codi vam poder implementar correctament aquesta opció.

Un altre problema que complicar una mica era quan es desconnectava un servidor o client. Concretament es devia al fet que nosaltres guardàvem la informació de tots els threads dels clients i els servidors en dos arrays, per tant quan un es desconnectava havíem de reajustar l'array i moure els elements tenint en compte que havia disminuït en 1 la quantitat. Per a solucionar el problema vam fer una funció que faria els shifts necessaris a les caselles de l'array per tal de redimensionar-lo. Però aquest shifts no van resultar fàcils, ja que alguns elements es perdien o es produïa una descompensació que ens petava

el programa. Un altre cop, amb algunes hores de debuggar i simplificant el codi al fer els arrays globals vam poder resoldre el problema.

Un dels problemes que ens va costar més resoldre, ens el vam trobar en intentar aturar els threads que s'havien iniciat per esperar rebre paquets dels clients que se'ns havien connectat o dels servidors als quals estaven connectats, un cop es desconnectaven. L'objectiu era aturar els threads iniciats per un usuari en concret en el moment d'una connexió. Per resoldre aquest problema, vam haver de crear un array que guardés la informació de cadascun dels threads que es creaven. Aquesta informació havia de contenir el nom de l'usuari a qui estava "escoltant" i la variable del thread. Amb això s'aconseguia tenir un control del threads que estaven en funcionament i, al rebre una petició de desconnexió, només caldria recórrer l'array buscant aquells creats per l'usuari que es desconnecta, i fer el join del seu thread. Finalment, en el switch que controlava els paquets que enviava el client, quan aquest tipus era de desconnexió vam implementar el següent codi per així resoldre el problema.

```
case 0x06:
    if(strcmp(p.header, "[]") == 0){
        //enviar el paquet de desconnexió OK
        enviaPaquet(c->sockfd, 0x06, "[CONOK]", 0, "");

        asprintf(&missatge, ADEU_CLIENT, c->user);
        escriuTerminal(missatge);
        free(missatge);
        imprimeixPrompt();

        //S'ha de modificar l'array de conn_serv
        //i restar una qServ;
        pthread_mutex_lock(&mtxC);
        eliminaConnexioCli(c->user);
        pthread_mutex_unlock(&mtxC);

        //aturem el thread
        connectatC = 0;

        close(c->sockfd);
        free(c->user);
        free(c);
    }
    else if(strcmp(p.header, "[CONOK]") == 0){
        connectatC = 0;

        close(c->sockfd);
        free(c->user);
        free(c);
```

```
}
break;
```

A l'opció de descarregar un àudio, es necessita obrir un nou fitxer on guardar els bytes que es van rebent. Vam tenir problemes en utilitzar la funció open amb l'opció d'escriure, ja que s'havien de passar més paràmetres que en el cas de llegir. Els paràmetres necessaris són el nom del fitxer, les opcions amb les quals s'obre el fitxer i els permisos que se li donen:

```
audioFile = open(p.data, O:WRONLY | O_CREAT | O_TRUNC, 0777);
```

En el cas dels àudios, vam decidir que el servidor enviaria la informació del fitxer en paquets de 512 bytes de data. El problema que ens vam trobar va ser que, majoritàriament, els àudios no tenien una mida total que fos múltiple de 512. Això provocava que l'últim paquet llegís més bytes dels que tenia el fitxer i, per tant, s'enviessin bytes erronis. La solució a aquest problema va ser llegir els bytes del fitxer original d'un a un fins a arribar a 512 o fins a acabar la lectura del fitxer, el que passés abans. Amb això s'aconsegueix que la mida de l'últim paquet fos d'una mida inferior a 512 però amb totes les dades correctes.

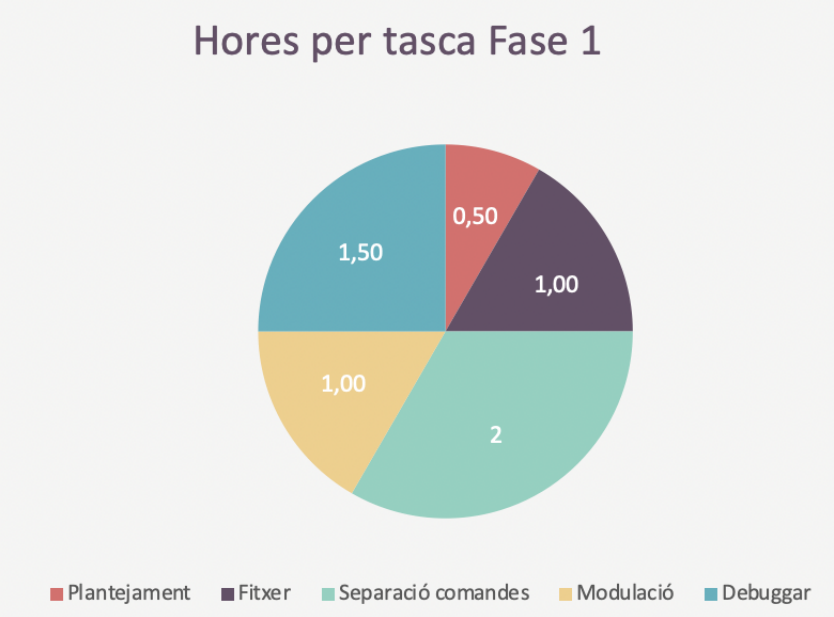
En tancar el Trinity, volíem aconseguir aturar el thread que estava a l'espera de noves connexions. Per fer-ho vam intentar aturar el bucle constant en el qual estava, però això no va funcionar, ja que dins el bucle hi havia la funció accept() que es manté bloquejada a l'espera de peticions. Això impedia que el bucle continués i, per tant, el thread no s'aturava. La solució va ser desbloquejar l'espera de connexions. Per fer-ho vam trobar que existia la funció shutdown(), que necessita rebre com a paràmetres el file descriptor del socket que accepta connexions i una constant que indica si es volen deshabilitar recepcions i/o transmissions. És important que no es tanqui el file descriptor abans d'executar la funció. Si no, aquesta no funciona.

3 Estimació temporal

3.1 Fase 1

Aquesta fase va ser la més senzilla i ràpida, ja que va ser la primera prova de contacte amb la pràctica. Només se'ns demanava processar el fitxer de configuració i separar les funcionalitats de les diferents opcions del menú.

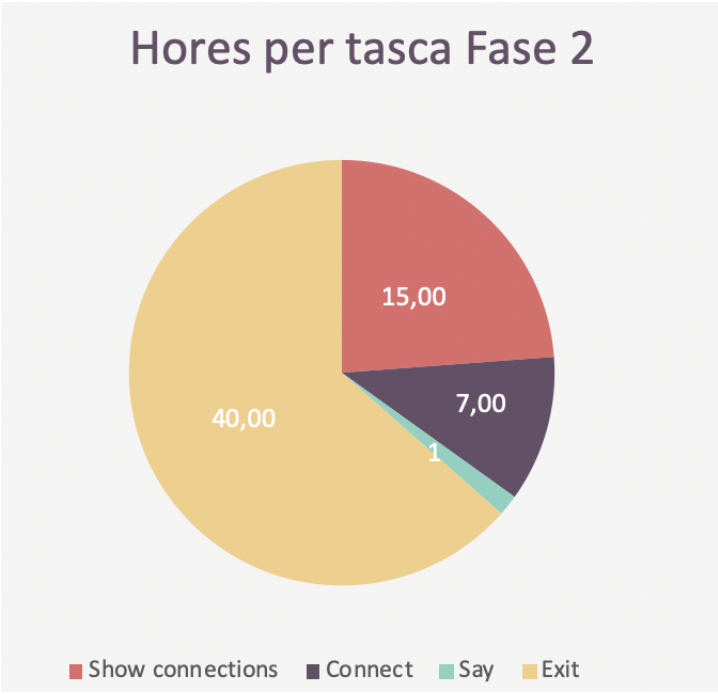
Podem observar que el plantejament ens va resultar bastant senzill. La lectura del fitxer i la modulació ens va portar aproximadament una hora cada tasca. La separació de comandes és la tasca que més temps ens va llevar, unes 2 hores aproximadament, per tal de tenir un codi més estructurat en un futur. Debuggar aquesta fase no va ser una feina difícil, ja que pràcticament no hi havia connexió entre els mòduls i no hi havia cap opció implementada.



3.2 Fase 2

En aquesta fase vam haver d'implementar les primeres opcions, en concret les opcions Show Connections, Connect, Say i Exit.

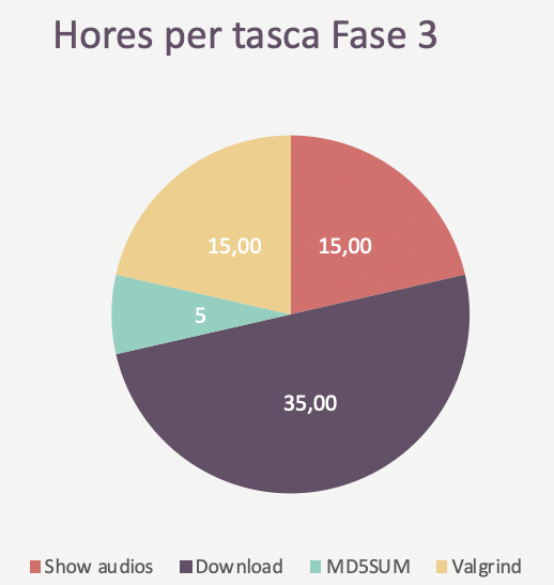
Podem veure que l'opció Show Connections ens va portar bastant de temps, sobretot per la interpretació del script, ja que ens donava diversos problemes al mac els quals no existien amb windows. L'opció connect ens va suposar unes 7 hores de treball, ja que havíem de tenir en compte tots els casos entre clients i servidors Trinity. L'opció Say ens va costar poc més d'una hora, ja que simplement es tractava d'un enviament d'informació a un socket concret. En aquesta fase ha sigut segurament la que més ens ha costat implementar una opció del menú, en aquest cas l'Exit. Dins de les 40 hores que hem calculat també entra el debuggar aquesta opció, ja que vam tindre diversos problemes i casos a considerar en desconnectar-nos o sortir del programa segons la quantitat o la connectivitat entre Trinitys.



3.3 Fase 3

En la tercera fase ja començàvem a transferir i descarregar arxius d'àudio, amb les opcions Show Audios i Download. A més, havíem de constatar l'intercanvi d'informació amb un checksum, així verificàvem que l'àudio s'havia descarregat correctament.

La implementació de Show Audios ens va suposar unes 15 hores, sobretot per la visualització de la llista d'àudios. L'opció download sí que ens va suposar molt de temps, fins a unes 35 hores, ja que vam tenir molts problemes una vegada l'arxiu estava descarregat, ja que no es podia reproduir o no tenia la durada de l'àudio inicial. Gràcies també a la implementació del checksum, que ens va portar unes 5 hores, vam poder resoldre aquests problemes i els arxius ja els descarregàvem sense cap problema. Finalment li vam dedicar unes 15 hores a debuggar sobretot en tema de memòria dinàmica, tasca per la qual va ser vital l'ús de l'eina valgrind, per trobar on havíem perdut paquets o emprat malament la memòria dinàmica.



3.4 Fase final

La fase final només afegia l'opció de Broadcast, però al ser l'entrega final havíem de revisar tot el codi a grans mesures, tant en tema d'eficiència, memòria dinàmica, claredat del codi, i tot el que requereix un programa ben fet.

L'opció que ens quedava en si no ens va portar molt de temps, de fet amb poc més d'una hora ja estava acabada, ja que es tractava en certa manera d'una extensió de l'opció Say que ja havíem implementat. Una vegada implementades totes les opcions només quedava retocar i millorar el codi. En tema d'optimització de memòria vam dedicar-li unes 10 hores, pel fet que ja li havíem dedicat temps en fases anteriors però encara quedaven detalls per ajustar sobretot en sortir del programa. Debuggar un altre cop és el que ens va fer dedicar-li més temps, més o menys el doble que amb el valgrind, a conseqüència de tots els mòduls i funcionalitats que contenia la nostra pràctica després de tot el semestre avançant en ella. Per últim, la memòria en tenir tants apartats i requerir una bona explicació perquè s'entengui el codi entregat, ens va suposar unes 15 hores de treball.

4 Conclusions i propostes de millora

En acabar la pràctica de Sistemes Operatius podem arribar a diverses conclusions.

La primera, i la que més ens ha obert els ulls, és que tot i haver fet dos anys d'informàtica, no sabíem programar com és degut en C. Aquest va ser el llenguatge que vam utilitzar a Programació I, però en segon el vam tocar molt tímidament. En arribar a aquesta assignatura, tant pel temari com per l'exigència de codi que es demana, hem après a programar com és degut en C. En haver programat molt amb Java i altres llenguatges que et simplifiquen molt la vida, no érem conscients de la cura que s'ha de tenir en estructurar i implementar codi en aquest llenguatge, sobretot pel fet d'alliberar recursos i optimitzar les eines i la memòria que emprem. A més, al no poder emprar l'eina Debugger que tant ens va ajudar en DPO hem après a saber detectar errors de codi, però sobretot a prevenir-los, ja que al saber que el programa ha de tenir una certa consistència i qualitat per fins i tot fer proves.

També hem de fer èmfasi en totes les eines i els coneixements apresos a classe, a més d'òbviament aplicar-los a la pràctica i veure tant l'eficàcia com la complexitat de la seva utilització. Tot i que hi havia moltes eines que ja coneixiem, al veure-les per primera en C i a l'haver de canviar la mentalitat a l'hora de programar ens ha enriquit molt com a programadors i ha ampliat en gran manera la quantitat de recursos que se'ns haurien de venir al cap a l'hora de resoldre problemes o plantejar projectes reals.

Aquesta assignatura ens ha servit per veure i ser conscients de la complexitat de fer un codi perfecte, o quasi perfecte. Ja no només pels requisits i l'exigència amb els warnings i la utilització de memòria dinàmica, sinó també pel fet que fer un codi ben estructurat i eficient inicialment ajuda molt i simplifica la feina pel futur. Ens hem pogut adonar per l'evolució que hem fet durant el semestre, amb els laboratoris, a la teoria i a la mateixa pràctica. Al principi programàvem d'una manera una mica innocent i acostumats a millorar el codi a base de prova i error. Mentre que al final del semestre hem tractat d'anar afegint el codi de manera més meditada i evitant errors potencials que poguessin sorgir.

Com a primera proposta de millora, ens semblaria interessant que hi hagués també una interfície gràfica, encara que fos senzilla. Això segurament motivaria més als alumnes i les execucions més visuals i fàcils d'entendre. A més, la implementació d'una interfície gràfica, si el codi està ben modulat, no hauria de suposar un gran inconvenient, ja que només caldria fer les vistes i amb un controlador passar-li els recursos necessaris per a cada vista.

També per fer el desenvolupament de la pràctica més entretingut estaria bé que els clients i servidors tinguessin alguna relació temàtica, com s'ha fet a

pràctiques anteriors que hi havia temàtiques o referències d'AC/DC, els Simpsons o Star Wars, entre altres. Un exemple interessant podria ser una temàtica política, que en els últims anys està donant bastant de què parlar. Amb referències a la independència, la incapacitat de formar govern o el ressorgiment de l'extrema dreta.

Una altra proposta que ja hem implementat a infinitud de pràctiques seria fer una espècie de rànquing o comptador d'estadístiques. Per exemple fer un rànquing dels àudios més descarregats d'un servidor, o els clients més actius, segons quants missatges han enviat o quants àudios han descarregat.

5 Bibliografia utilitzada

Documentation. New to Latex? *Overleaf* [en línea]. [Consultat el 19 de maig de 2019]. Disponible en: https://es.overleaf.com/learn

LlibreUNIX 14-15. *eStudy* [en línea]. [Consultat el 28 de octubre de 2019]. Disponible en: https://estudy.salle.url.edu/pluginfile.php/792145/mod_folder/content/0/LlibreUNIX_14-15.pdf?forcedownload=1

Annex protocol comunicació. *eStudy* [en línea]. [Consultat el 10 de novembre de 2019]. Disponible en: https://estudy.salle.url.edu/pluginfile.php/833825/mod_resource/content/2/Annex_1_Protocol_comunicacio%20v1.2.pdf

shutdown(3) - Linux man page. *die.net* [en línea]. [Consultat el 24 de novembre de 2019]. Disponible en: https://linux.die.net/man/3/shutdown

Valgrind Invalid free(). *Stackoverflow* [en línea]. [Consultat el 8 de desembre de 2019]. Disponible en: https://stackoverflow.com/questions/36815937/valgrind-invalid-free-delete-delete-realloc-in-c

fork() in C. *Geeksforgeeks* [en línea]. [Consultat el 2 de novembre de 2019]. Disponible en: https://www.geeksforgeeks.org/fork-system-call/

PTHREAD_MUTEX. *skrenta* [en línea]. [Consultat el 14 de novembre de 2019]. Disponible en: http://www.skrenta.com/rt/man/pthread_mutex_init.3.html

dup(2). *Linux manual pages* [en línea]. [Consultat el 19 de novembre de 2019]. Disponible en: http://man7.org/linux/man-pages/man2/dup.2.html

execvp(3). *Linux manual pages* [en línea]. [Consultat el 8 de novembre de 2019]. Disponible en: https://linux.die.net/man/3/execvp

readdir(3). *Linux manual pages* [en línea]. [Consultat el 3 de desembre de 2019]. Disponible en: http://man7.org/linux/man-pages/man3/readdir.3.html